Communication in Worldwide Distributed Object Systems

Henry Detmold, B.Sc. (Hons.)

A Thesis Submitted for the Degree of Doctor of Philosophy

Department of Computer Science

The University of Adelaide

March, 2000

Table of Contents

List of Figures

List of Tables

List of Equations

Abstract

The construction of distributed systems on the worldwide scale is a challenging problem. Given fundamental physical limits, the latency of communication in such systems will always be large, so the most pressing challenge is to find ways to minimise the effect of this latency on system performance. However, it is important that facilitation of software construction not be sacrificed in an all out assault on the latency challenge, so a secondary challenge is to provide a good programming model for such systems. Finally, most programmers learn their craft in localised systems, so it important that this programming model be as familiar as possible. These challenges intersect in the notion of programming language level communication mechanism; the subject of this thesis.

The first part of this thesis presents a model for programming wide area distributed systems. This is based on the object oriented programming model as this is the most appropriate programming paradigm for many distributed systems. The object model employed in this thesis is underpinned by notions of software contracts that are intrinsically part of objects and that govern both the use and the implementation of those objects.

The second part of the thesis develops a framework for assessment, analysis and comparison of communication mechanisms in worldwide distributed object systems. This framework is applied to the analysis of a number of existing mechanisms.

The third and final part of this thesis presents three new communication mechanisms: *Entitlements*, *Responsibilities* and *Ambassadors*. Each of these constructs takes a slight different approach to the thesis subject, and is successful, to different degrees, in meeting the challenges motivating the work.

Declaration

This work contains no material which has been accepted for the award of any other degree or diploma in any university or any other tertiary institution and, to the best of my knowledge, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my Thesis, when deposited in the University Library, being available for loan and photocopying.

Henry Detmold, Thursday, 2 March 2000.

Acknowledgments

I gratefully acknowledge all the help and encouragement I have received during my candidature as a Ph.D. student. I am indebted to my supervisor, Dr. Michael Oudshoorn, for his guidance, thoughts and encouragement during my candidature. I am thankful for the patience shown by my wife, Larissa, and by the other members of my family whilst I undertook this work.

I thank the Department of Computer Science generally and the head of department, Professor Chris. Barter, in particular, for providing a productive research environment. Specific thanks go Michael Hollfelder, who undertook aspects of the work noted in Chapter 8 during his Honours candidature, to Dr Ken Hawick of the Distributed High Performance Computing centre at Adelaide and the Australian National University in Canberra for providing access to equipment on which to conduct the experiments in Chapter 8, and to Professor Jean-Luc Gaudiot at the University of Southern California for providing access to equipment similarly.

Chapter One

Introduction

This thesis develops high-level support for the communication facilities required by distributed systems deployed on the global scale. Existing high-level distributed systems communication technologies have typically been developed for local area network deployment, and consequently perform poorly when translated to the high-latency environment of distributed systems on the worldwide scale. The work described in this thesis is an attempt to confront the challenge posed by this high latency environment.

This work is targeted at providing high-level support for communication; that is, support at the level of programming languages and above. As such, it needs to be undertaken within the context of a programming paradigm. The choice made is for a general object-oriented model. The rationale for this choice is that it is the programming paradigm that most closely fits the task of modelling distributed system components.

Distributed systems of worldwide scope typically must integrate existing (legacy) components implemented in diverse programming languages. Furthermore, they must execute on a constantly evolving variety of hardware architectures and operating systems. These are all requirements for heterogeneity, which the work described in this thesis must support.

Finally, systems on the worldwide scale often must operate across multiple administrative and security domains, with no single domain in overall control. The work described in this thesis must ensure compatibility with techniques emerging to address it.

1.1 Motivation

High-level distributed systems mechanisms, such as remote procedure call (RPC) [Birell and Nelson 1984, Nelson 1981] in distributed imperative programming, and remote method invocation [Sun 1997a] in distributed object-oriented programming, were originally designed for local area network (LAN) environments. Although these mechanisms continue to function on the world-wide scale, the performance they offer on that scale falls short of that required for the next generation of distributed systems.

The problem results from the vast increase in magnitude of the latency on the worldwide scale in comparison to the LAN scale. A typical LAN, servicing a department of a corporation, might have a maximum diameter of 300 metres. In theory, a message can propagate across this LAN in one microsecond. In practice, inefficiencies in current operating systems and (to a diminishing extent) in network hardware combine to increase the latency towards one millisecond, but the theoretical minimum is capable of being approached in research systems [Rodrigues, Anderson and Culler 1997]. In contrast to the LAN, the typical worldwide distributed system would have a diameter the order of of 30,000,000 metres. In this network, the minimum transmission latency is 100 milliseconds. This is without accounting for switching delays and assumes the necessary bandwidth to carry the message is immediately available. Consideration of these factors raises the average latency (for small messages) to over half a second. A remote procedure call requires a round trip of two messages, so the delay experienced in making a world-wide RPC is of the order of one second, plus whatever time is taken for the server to actually execute the call. With such high latencies, it is clearly unsatisfactory to adopt the LAN-scale approach of issuing an RPC and then blocking, idly awaiting the reply containing its result.

One way to confront this challenge is entirely to dispense with the RPC/RMI technology, instead resorting to lower level mechanisms such as message queues or even direct use of transport protocols. This, however, loses many of the benefits for which RPC/RMI was first adopted. Such a retrograde step should therefore be avoided. The proper response to the challenge is to address the sensitivity to latency of conventional RPC/RMI whilst preserving (or in some way matching) its high-level programming paradigm. This thesis adopts the distributed object-oriented paradigm supported by RMI as the programming paradigm.

RPC/RMI have also proven effective in supporting heterogeneity in LANs. The need for heterogeneity is even more pressing in worldwide systems, so a high-level mechanism for worldwide communication must provide support to an equivalent level.

1.2 Context, Focus and Limitations of Inquiry

Two requirements distinguish the development of distributed systems from the development of centralised systems:

- The requirement to communicate between different nodes in the system.
- The requirement to manage independent failures of different nodes and of the network.

The mechanism for communication must provide some level of resiliency against failures and it must provide some support for the detection of failures against which it is not resilient. Therefore, the communication mechanism is the more basic of the two requirements. Hence, the focus of this thesis is on the communication mechanism(s) used in distributed systems.

1.2.1 The Rise of Distributed Systems

One way to categorise distributed systems is according to the factors leading to their existence. A given distributed system will have developed according to one of two models:

- The ***intrinsic distribution*** model is applicable when the function of the system requires physical distribution of the hardware executing the system.
- The ***discretionary distribution*** model is applicable when there is no such requirement for physical distribution, but the system is nevertheless distributed for increased performance and other reasons.

This list of categories is exhaustive for all possible distributed systems. The only other possibility – where a system is neither required to be distributed by physical factors, nor is it distributed for discretionary reasons – is of course a centralised rather than a distributed system.

The primary factor leading to systems under the intrinsic distribution model is the interaction of the computation with the real world. A system with only one point of real-world interaction may be implemented as a centralised system on the computer that provides the point of interaction. A system in which real world interaction occurs at several distinct and autonomous computers must be subject to mandatory distribution: at the very least, it must be a system distributed across the computers attached to each point of interaction. In practice, the most common cause of multiple points of real-world interaction is the existence of multiple human users of a system. Another cause might be scarcity of specialised hardware, *eg.* in most Unix local area networks, only a minority of computers are directly connected to a printer; the others must access the available printers via a simple distributed system.

Discretionary distribution arises for various pragmatic reasons. Suppose one has a computationally intensive task in the domain of mathematical or scientific computing. This system has a single point of real-world interaction, obtaining initial data from the user and reporting the final results to that

Intrinsic Distribution	Discretionary Distribution
• Distributed file systems. • Collaborative desktop productivity applications, such as word processors which allow multiple users to work simultaneously on (different parts of) a single document. • Communication applications, such as E-mail and network news tools. • Resource discovery and retrieval applications, such as World Wide Web browsers and automated search engines. • Enterprise database management systems, providing enterprise-wide access to data produced at multiple locations within an enterprise. • On-line transaction processing (OLTP) systems, such as airline booking systems and automatic teller machine networks. • Distributed education systems.	• Computationally intensive mathematical and scientific programs. • Batch processing systems.

Table 1 – Examples of Intrinsic and Discretionary Forms of Distribution.

same user. If the only processing capability available is one or more networked workstations, it may be desirable to partition the system to simultaneously employ the computing resources of several of these workstations in order to reduce the elapsed time of the system in solving a particular problem. This is an archetypal discretionary distributed system: even though it could be implemented as a centralised system, it is implemented as a distributed system instead in order to deliver some benefit. In this case the benefit is reduced elapsed time for the system to produce results for a given problem.

In the case of discretionary distributed systems, there is always a question of whether a centralised version of the system will run faster. The reason that the centralised system may be faster is that the cost of communication in the distributed version may outweigh the increased computational capability. The question of whether to distribute does not arise with intrinsic distribution: the system must be distributed!

1.2.2 Focus on Intrinsically distributed Systems

In local area networks, the latencies of communication operations are of the order of a few milliseconds (and are potentially of the order of a few tens of microseconds). Discretionary distribution is often a wise choice in such networks, because the increased cost due to communication can be offset by the availability of increased processing power.

In worldwide networks, the cost of communication is at least several hundreds of times greater[1] than is the case on LANs. For a discretionary distribution to be a wise choice in this environment, the increase in computational capacity must also be hundreds to thousands of times greater. It is obvious that discretionary distribution is far less often a viable choice in worldwide systems than it is in local area network systems.

In consequence, this thesis focuses only on intrinsically distributed systems. As can be seen by inspection of Table 1, this is not overly restrictive in scope of application: the category includes a large variety of interesting systems. In any case, the technology examined and developed in this thesis, whilst specialised for the requirements of intrinsically distributed systems, is also applicable to discretionary distributed systems.

[1] Comparing a 3km LAN with a 30,000 km WAN, the theoretical ratio of latencies is 1:10,000. In practice, it is of the order of several hundreds to a few thousands.

1.2.3 Communication Paradigms for Distributed Computing

One approach to supporting distributed systems is to abstract over distribution completely. *Distributed shared memory* (DSM) systems provide communication through the mutation of shared data, exactly as in centralised systems.

Currently, the most common form of distributed system in the commercial world consists of clients communicating via interaction with database servers. This option is termed *distributed database access*. Since all interactions between system components occur via database servers, this paradigm for interaction is somewhat inflexible.

The final paradigm considered is explicit *inter-process message passing*, in which the sending of a (request) message by a thread in one address space triggers the execution of an operation by a different thread in a different address space. This paradigm allows communication between arbitrary nodes in the system and is hence inherently more flexible than is distributed database access. Message passing systems can be augmented with *TP (transaction processing[2]) monitors*, sometimes called *message-oriented middleware*. TP monitors add transaction properties and protection to message passing distributed systems whilst permitting them to retain their inherent flexibility.

1.2.3.1 Distributed Shared Memory

Distributed shared memory systems simulate the facilities offered by shared memory multi-processors. Threads anywhere on the network may share memory, and communicate via operations on this shared memory.

Some of the most common examples of modern (centralised) shared memory systems are multi-threaded Unix processes. All threads within a Unix process execute in the same context, against the same address space with the same memory protection environment. It is possible to construct an analogous model with distributed shared memory.

An improvement on the basic sharing functionality is to take further advantage of the memory protection and address translation in order to increase both safety, through memory protection, and flexibility, through mapping. Several facilities are desirable:

- Threads may have differing memory protection environments.
- Memory may be private to a single thread, shared between several threads, or globally shared.
- Address spaces may be constructed by mapping of parts of other address spaces.

The Grasshopper operating system [Rosenberg *et al.* 1996] provides these kinds of facilities.

Implementation

Distributed shared memory is usually implemented at the page level. Machines in the distributed system cache copies of the pages in the distributed address space in local physical memory. A coherency protocol is used to maintain consistency between the caches. The most common form of consistency requires that pages exhibit *one copy semantics*, that is, every access to a copy of a shared page behaves as if it were an access to a single physical page.

1.2.3.2 Distributed Database Access

Client programs access a database via the execution of *transactions*. Distributed database access permits clients to execute on different machines to the machine(s) which host the database server(s) they use, thereby creating a distributed system.

---

[2] The TP acronym originally stood for Tele-processing, but its use to mean transaction processing is now almost universal.

Transactions [Eswaran *et.al.* 1976,Gray and Reuter 1992] (whether in a distributed or a centralised environment) have the so-called *ACID* properties listed below:

- ***Atomicity*** – with regard to updating the database, transactions appear to execute to completion or not at all. It is not possible for another transaction to observe the effect of a partially completed transaction in the database. Transactions that execute to completion are said to *commit*. Transactions that do not execute to completion are said to *abort*. Transaction aborts may be requested by the program running the transaction, may be caused by the transaction processing system or may be the result of failure of some part of the system. In any of these cases, there is no effect on the database.
- ***Consistency*** – any transaction, when executed in isolation and to completion, transforms the database from a consistent state to another consistent state. This is a property of the operations executed by the transaction, and hence is under programmer control.
- ***Isolation*** – transactions appear indivisible to each other: a collection of transactions executed concurrently has the same effect as if that collection of transactions were executed sequentially in some order.
- ***Durability*** – the effects of (committed) transactions against the database have a very high probability of surviving future failures.

In centralised systems, transactions provide an advantage over shared memory because they raise the level of abstraction of (atomic) actions against the shared store from hardware reads and writes to some higher level determined by the system design. Hence, programmers do not have to worry about interactions between machine instructions executed by different threads in the system, only about interactions between the much smaller number of transactions executed by these threads. The programmers' task is as follows:

1. Define invariants describing consistent states for the database.
2. Ensure that all transactions individually exhibit the consistency property in preserving those invariants.

A second advantage of transaction processing systems over most shared memory systems is that developers do not need to address the issues of failure and durability.

Implementation

In most current distributed database access environments, such as Oracle [Oracle 1997], clients communicate with the server via the use of a query language such as SQL [ISO 1992]. Transactions consist in a series of SQL statements, possibly with interposed client code. The client sends SQL statements to the server, and the server interprets and executes them. Typically, the majority of the execution of a transaction is performed at the server end, with the client mainly being responsible for overall control and the user interface. Furthermore, modern database management systems (DBMSs) allow procedures to be sent to the server, and then to be executed on the server in response to subsequent client requests, further increasing the amount of the application executing inside the server.

1.2.3.3 Inter-Process Message Passing

The explicit sending of messages between processes effects communication in the inter-process message passing paradigm. Processes receive messages and react to them, possibly by sending additional messages. Communication is often structured so that request messages are matched by corresponding reply messages. This structure is most often supported within a (remote) procedure call paradigm.

Processes in a message passing system accept a fixed number of kinds of messages. Each kind of message causing a different reaction when received and having an identifying name which is unique for that (recipient) process. Furthermore, messages of a given kind may be permitted to have parameters, the number and type of which is the same for all instances of the message kind. The actual values of the message parameters are set by the sender of the message and may effect the reaction produced in response by the recipient. In an extreme case, the message parameters might encode the operations to be executed, for example, the parameter to an *execute_SQL* message might be an SQL query of arbitrary complexity. Hence, one can see that distributed database access systems are message-passing systems with a special language (typically SQL) being sent in the messages.

Implementation

Processes have conceptually separate address spaces, therefore data must generally be copied rather than being passed by reference, however, the implementation of message passing between processes on the same hardware node may be optimised by the use of shared memory.

1.2.3.4 TP Monitors

As mentioned above, TP monitor systems such as Camelot and Avalon [Eppinger, Mummert and Spector 1991] and commercial derivatives such as Transarc's Encina (now owned by IBM), add transaction facilities to inter-process message passing. In a TP monitor system, services are registered with the monitor, which then treats each such service as a resource manager supporting ACID transactions on an entity of some kind.

Whereas in distributed database access there is only one kind of resource manager (the DBMS server), TP monitors allow other sub-systems to be treated as resource managers. In particular, the message passing system is treated as a resource manager. This has two consequences:

- Sequences of messages can be executed under transaction protection.
- The message passing mechanism is orthogonal to the provision of transaction services, provided it follows a specified protocol for interacting with the TP monitor.

In the distributed database access paradigm, transactions govern the interaction between a client and one (or more) database server(s). With TP monitors, transaction protection can be provided to a significantly more flexible range of activities, some examples are discussed below:

- Access to a work queue. Since the queue is recoverable, this means transactions can place items onto the queue and then commit before the items are removed and operated on (by some other transaction). This strategy can prove quite effective in reducing the duration of transactions.
- So called *conversational interactions*. These consist in the exchange of an arbitrary number of messages between components of the distributed system. Conversational interactions between two components are by far the most important case, but conversational interactions involving more than two components are also possible.
- *Work-flows*. These consist of scripts that send information between multiple applications and users for processing and revision; for example, the process of authoring, reviewing, annotating and revising a document prior to publication.

Implementation

A TP monitor implementation has three main sub-systems:

- A transaction engine, responsible for ensuring that the activities in the monitored system exhibit the desired (transactional) semantics. This is concerned with the management of failures, not with communication, and is therefore beyond the scope of this thesis.
- A process scheduler and allocator, responsible for starting and stopping server processes to optimise use of resources in the distributed system. This capability is not really a core part of

TP monitor task, but is typically included in TP-monitor products. In any case, it is also beyond the scope of this thesis.

- A message transport sub-system.

The message transport element of a TP monitor system is orthogonal to the transaction-processing element, it is possible to use a standard message passing mechanism for the message transport. For example, message transport in the *Encina* TP monitor is performed by the Remote Procedure Call (RPC) mechanism from the Open Software Foundation (OSF) Distributed Computing Environment (DCE) [OSF 1992].

1.2.3.5 Comparison of Distributed Communication Paradigms

Message passing systems running under the supervision of a TP monitor generalise and extend the transactional facilities available through distributed database access. Consequently, it is not necessary to consider further the distributed database access paradigm. In addition, since the transaction processing mechanism of message passing TP monitor systems is orthogonal to the (message passing) communication mechanism, it is not necessary to consider message passing under a TP-monitor as distinct from message passing alone. Finally, research systems such as Clouds [Dasgupta *et.al.* 1990] demonstrate that DSM can run under a transaction processing regime, so the choice between DSM and message passing is orthogonal to any requirement for transaction processing. Consequently, this comparison involves only message passing and distributed shared memory and ignores transaction-processing issues completely.

The choice of the best paradigm is generally determined by the application system. In fact, some applications may best be served by a combination of both DSM and message passing, possibly under the supervision of a TP-monitor. The decision can be informed by consideration of the following factors:

- Performance.
- Concurrency control mechanisms.
- Ease-of-understanding.
- Ease-of-use.
- Requirement for linguistic heterogeneity.

Each of these is discussed below.

Performance

Distributed shared memory systems perform poorly when threads on different processors access data on the same page and at least one of the threads is mutating the data. This can occur for two reasons:

- The threads are all trying to access the same data.
- The threads are trying to access several disjoint data elements that are coincidentally located on the same page (this is known as *false thrashing*).

The problem occurs because any update to a shared page necessitates all cached copies of that page on other nodes to be invalidated. For the next access from any of these nodes, whether it is a read or a write access, the page must be re-loaded over the network.

This is similar to the problems of cache-line thrashing in shared-memory multi-processors, but is even more severe in degree. To counter the false-thrashing aspect of the problem, it may be possible to insert padding to align data on page boundaries. Obviously this is only a reasonable strategy for data of a size similar to or greater than the page size. It may also be possible to re-structure programs to avoid genuine thrashing. For example, a system in which several threads repeatedly add to a single shared variable may be restructured so that each thread adds to a local (unshared) variable repeatedly and then (only once) adds the final value of the local variable to the shared variable. Finally, the extent to which the problem occurs depends on the interaction between the application and the memory coherence protocol used by the DSM. Different combinations may yield radically different performance and systems that include a facility for applications to request particular memory coherence semantics may support a general improvement.

The performance of early transaction processing systems is poor. This derives from the high cost of (disk) I/O and from low degrees of concurrency. However, because of the commercial importance of database applications, a great deal of effort has been expended in the improvement of these aspects. Modern systems are able to avoid almost all synchronous I/O and to perform almost all disk writes sequentially (thereby avoiding the overhead of random access seeks). Using advanced concurrency control strategies, the degree of concurrency has also been dramatically raised.

DSM and distributed transactions are both built on top of message passing facilities, so it will always be possible to obtain equivalent or superior performance using explicit message passing. The drawback is that it is not always straightforward to achieve.

Concurrency Control

Distributed shared memory systems require explicit concurrency control mechanisms such as semaphores or monitors. Furthermore, if concurrency control on regions of memory smaller than a page is supported, the semantics of the concurrency control can be contaminated by the page structure. The result of the contamination is to be more pessimistic than that of the concurrency control primitive in a physically shared memory environment.

Consider a page containing two independent buffers, one locked for write by a remote process and one unlocked. According to the semantics of the concurrency control primitive, it should be possible to apply a write-lock to the unlocked buffer and proceed to modify it immediately. However, because the DSM system will only permit a single node to have a modifiable copy of the page containing the buffers, the process will have to wait for the lock. In itself, this reduces the amount of concurrency, and hence performance, but does not alter the semantics of concurrency control. An even worse situation occurs where a process tries to acquire a lock that *is* held by a remote process. In this case, the page containing the locked buffer will be removed from the remote node and installed into the local node. Only then will the local process find that the lock is held and hence it has no use for the page it has just acquired. The page will eventually make its way back to the remote node to be updated by the process holding the lock. In effect, the page has made a round trip across the network with no useful purpose served. To overcome this problem the DSM system can interact with the concurrency control so that copies of pages containing locks are not released until the locks are released. This means the mutations of shared buffers are artificially serialised at page granularity, and this is a contamination of the semantics of the concurrency control mechanism. This artificial serialisation can lead to deadlocks which are not present in the uncontaminated system, and which are very difficult to diagnose and to correct.

In contrast, with message passing there is no sharing (at least none visible to the programmer) and therefore no need for concurrency control. In a message passing application, synchronisation occurs as part of communication. Therefore, safety properties such as deadlock freedom can be discerned from the code of the application – in contrast to the DSM case there is no contamination of these properties.

Ease of Understanding

Concurrent systems which update a shared store in a cooperative fashion[3] are generally more difficult to understand and reason about than are message passing concurrent systems. In particular, the extant

[3] Co-operative concurrency, in which the updates made by any thread are immediately visible to all threads. This is in contrast to transaction based concurrency, in which each thread of computation is isolated from the others.

formalisms for describing and reasoning about concurrency, such as CCS [Milner 1989], CSP [Hoare 78,Hoare 85], Actors [Agha 1986], and π-calculus [Milner 1993], deal with message passing between sequential processes.

DSM-based systems involve the concurrent update of a shared store in a cooperative fashion. Consequently, applications that require complex communication patterns between threads suffer from poor understandability if implemented using DSM. In such cases, explicit message passing is usually to be preferred.

Ease of Use

Distributed shared memory is very effective as a simple means to distribute information to the nodes of a distributed system. Required information is brought into the local physical memory cache automatically as it is used and is accessed in exactly the same way as memory in a centralised system. DSM becomes difficult to use when it is necessary to restructure application code to counter performance problems due to page thrashing or due to contamination of concurrency control semantics.

Message passing is relatively cumbersome for the simple data distribution and partitioning tasks where DSM excels. For systems exhibiting complex patterns of communication, message passing is no more difficult to use.

Requirement for Linguistic Heterogeneity

As is discussed in detail in Chapter Two, there are several kinds of heterogeneity that a distributed system might support. Linguistic heterogeneity is the ability for the computational elements of the system to be implemented in a variety of programming languages. That is, the computational elements of a linguistically heterogeneous distributed system may be implemented in several different programming languages, each with access to a shared means of communication.

Supporting linguistic heterogeneity raises additional challenges to the implementation of a DSM system. In a *homogenous* DSM system, all (valid) copies of a given memory page in the DSM address space are physically identical. Given that this is the case, servicing a page fault simply requires location of a valid copy of the desired page and its copying over the network.

In a heterogenous system, different implementation languages might represent a given logical object with differing physical representations. This has several results, including:

- A collection of logical objects that fit into a page on one component of the system may not so fit on another component (implemented in a different language). Consequently, a given virtual memory page may have radically different logical contents at different components.
- Loading a page is no longer simply a matter of copying its contents, instead, a complex data format translation may have to be applied.
- Components on the same machine may not be able to share physical pages of the DSM if they are implemented in different languages. This means there may be several physical copies of a given logical object in memory at that machine.

These complications lead to great increases in implementation complexity and to a significant reduction of the performance of that implementation. Given that even linguistically homogenous DSM is widely considered to deliver inadequate performance, it seems unlikely that linguistically heterogenous DSM can deliver performance suitable for a general purpose communication mechanism.

Applications Appropriate for DSM

The strengths of DSM lie in homogeneous systems which leverage the ease-of-use benefits described above whilst avoiding the potential for poor performance, poor understandability and contamination

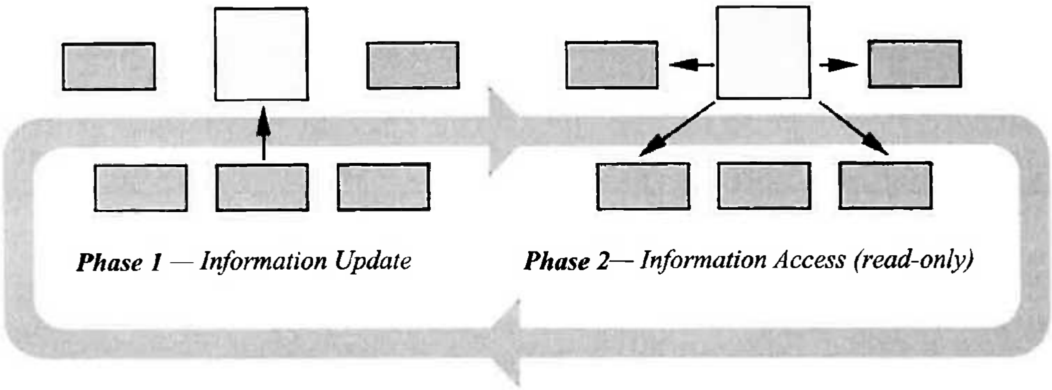

Figure 1 – Distribution of Infrequently Modified Information.

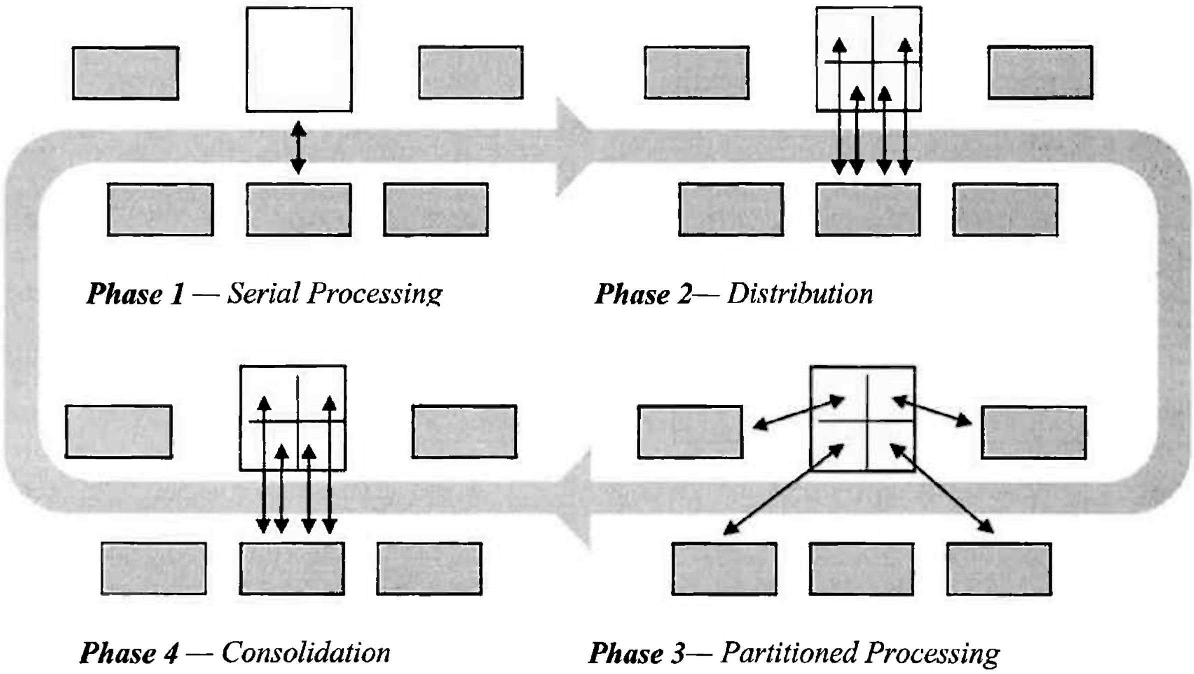

Figure 2 – Distributed Access to Partitioned Data.

of concurrency control also described above. There are two main kinds of applications where it is possible to attain these characteristics:

- Applications involving distribution of infrequently modified information.
- Applications involving distributed access to disjoint partitions of data.

Applications requiring distribution of infrequently modified data operate in two phases as shown in Figure 1. First, a single thread in the system publishes or updates the information. Secondly, multiple threads read (or subscribe to) the information, causing it to be loaded into the page caches on the threads' nodes. These phases might be repeated in a cyclical fashion as shown in the figure; in order to obtain satisfactory performance, the duration of the subscription phase needs to be several orders of magnitude longer than that of the publication phase.

Some concrete examples of this kind of access characteristic include the distribution of libraries of object code, the distribution of documentation and distribution of other *infrequently updated databases* such as spelling dictionaries, system configuration information and standards definitions.

Applications exhibiting partitioned access to data operate in four phases as shown in Figure 2. In the first phase, a single (master) thread processes data in a serial fashion. In the second phase, the master thread prepares the data for distribution to several (slave) threads. In the third phase, the slave threads process disjoint partitions of the data. Finally, in the fourth phase, the master thread consolidates the data in the partition to prepare for further serial processing. Again, the phases may be repeated in a cyclical fashion, and the level of performance obtained is determined by the duration of the third phase relative to the others.

Concrete examples of this kind of access exist mostly in the mathematical and scientific computing domains. In particular, many matrix-processing tasks can be structured in this way. It should be noted that distribution in the majority of such applications is discretionary rather than intrinsic in nature. As noted in Section 1.2.1 previously, this thesis ignores such applications.

Applications Appropriate for Message passing

Distributed shared memory is the more appropriate paradigm only in a small subclass of applications. Furthermore, many of these applications are outside the domain of intrinsically distributed systems on which this thesis focuses. The remainder of the field, including the majority of intrinsically distributed systems, is more appropriately served by message passing.

1.2.4 Focus: Inter-Process Message Passing

The remainder of this thesis deals exclusively with systems based on explicit inter-process message passing. The state of the art in message passing communication is explored in the next chapter.

1.3 Thesis Outline and Identification of Major Contributions

This thesis contains nine chapters including this introduction, the remaining chapters are divided into three parts. The first part explores existing work forming the basis for the major contributions of the thesis, as follows:

Chapter Two	An overview of distributed computing with message passing communication.
Chapter Three	A general overview of state-of-the-art of object technology in sequential systems.
Chapter Four	Discussion of concurrent and (in particular) distributed object systems.

The second part of the thesis contributes a novel framework for the analysis of communication mechanisms in distributed object based systems, providing an improved ability to analyse and understand existing work in the field:

Chapter Five	Development of a framework for the analysis of communication mechanisms in distributed object based systems according to software engineering, application and performance criteria.
Chapter Six	The application of the framework developed in the previous chapter to diverse communication mechanisms in the extant literature.

The third and final part of the thesis introduces as novel contributions three communication mechanisms: *Responsibilities*, *Ambassadors* and (novel to a lesser extent) *Entitlements*. The conclusion explores further development of these mechanisms beyond the scope of the thesis research.

Chapter Seven	Description of two language-level mechanisms for communication in worldwide distributed systems: *Entitlements* and *Responsibilities*.
Chapter Eight	The use of mobile Java objects as *Ambassadors* to avoid unnecessary communication latency in worldwide distributed systems.
Chapter Nine	Conclusions and future directions.

Chapter Two

Message Passing Communication Mechanisms

This chapter explores the basic mechanisms and methods used in the construction of message-passing distributed systems. Before exploring the mechanisms, it is necessary to explore some of the fundamental properties and limitations of the communication protocols supporting these mechanisms. The mechanisms in common use are then explored. A focus on the most generally useful kind of mechanism, remote operations, is then enunciated. Having identified this focus, which is further pursued in the remainder of the thesis, the discussion shifts to consideration of pertinent software engineering issues and then to heterogeneity. The chapter concludes by establishing the desirability of programming language support for message passing communication mechanisms, this topic is expanded upon in the next chapter.

2.1 Model of Computation in Distributed Systems

The section describes a simple model of computation in distributed systems and then proceeds to discuss the impact this model has on transport protocols supporting communication mechanisms.

2.1.1 Model of Computation

Following [Hadzilacos and Toueg 1993], computation in message passing distributed systems can be described in terms of the following parameters:

- Synchrony of processes and communication.
- Types of communication failures.
- Types of process failures.
- Network topology.
- Deterministic versus non-deterministic processes.

These parameters are described in Sections 2.1.1.1 to 2.1.1.5. The particular parameters adopted in this thesis are given in Section **Error! Reference source not found.**.

2.1.1.1 Synchrony of Processes and Communication

Distributed systems may be classified as either synchronous or asynchronous. Hadzilocos and Toueg define synchronous systems as those which satisfy the following properties:

- *There is a known upper bound δ on message delay; this consists of the time it takes for sending, transporting, and receiving a message over a link.*

- *Every process p has a local clock C_p with known bounded rate of drift $\rho \geq 0$ with respect to real time. That is, for all p and all t > t':*

$$(1+\rho)^{-1} \leq \frac{C_p(t)-C_p(t')}{(t-t')} \leq (1+\rho)$$

Where $C_p(t)$ is the reading of C_p at real time t.

- *There are known upper and lower bounds on the time required by a process to execute a step* [in the algorithm executed by the system].

Systems that do not satisfy all of the above properties are said to be asynchronous. In fact, the asynchronous category includes all synchronous systems as well, so this category is completely general.

2.1.1.2 Communication Failures

Typical models of process failures include the following:

- **Crash Failures** – a link can fail by ceasing to transmit messages. Prior to doing so, it behaves correctly.
- **Omission Failures** – a link can fail by intermittently losing messages.
- **Arbitrary (also called Byzantine or malicious) Failures** – a link can fail by exhibiting arbitrary behaviour. This includes, but is not limited to, the following: delivery of corrupt messages, re-ordering of messages, generation of bogus messages, crash and omission failures.

Each of these model subsumes those preceding it.

2.1.1.3 Process Failures

Typical models of process failures include the following:

- **Crash** – a process can fail by halting execution. Prior to doing so, it operates correctly. After doing so, it does not resume. There is no means for other processes to detect that a process has failed.
- **Receive Omission** – a process can fail by crashing (as above) or by intermittently failing to receive messages sent to it.
- **Send Omission** – a process can fail by crashing (as above) or by intermittently failing to send messages that it is meant to send.
- **General Omission** – a process can fail by a combination of send omission and receive omission.
- **Arbitrary Failures** – a process can fail by exhibiting arbitrary behaviour, including collusion with other failed processes.

With the exception of send omission, each of these models subsumes those above it. Send omission, like receive omission, subsumes crash.

2.1.1.4 Network Topology

Important considerations of network topology include:

- Whether communication is broadcast or point-to-point.
- Whether the network can partition, and the characteristics of such partitions.

2.1.1.5 Determinism versus Non-determinism of Processes

Systems in which processes may operate in a non-deterministic way are more difficult to model than are systems in which all processes are deterministic.

2.1.1.6 The Model of Distributed Computation in this Thesis

2.1.1.6.1 Assumptions about Synchrony of Processes and Communication

In practice, distributed systems need to be modelled as asynchronous systems. This is because practical aspects of real computing environments, such as the widespread use of time-sharing operating systems and network technologies with probabilistic behaviour (eg. Ethernet), violate the synchrony assumptions. This practical view is the one taken here: namely that distributed systems are asynchronous.

However, it is possible to have some of the synchrony properties and not others. It is also possible to use approximations to some of these synchrony properties. A *failure detector* is an abstraction that allows a process within an asynchronous system to discover information about whether another process in that system has crashed. A failure detector, which is denoted *failed*(*remote-process*), is a local function which allows a (local) process to obtain information about some remote process (the *remote-process* parameter). The two possible results of the function are interpreted as follows:

- **Positive** – *remote-process* had definitely not crashed when the failure detector was queried.
- **Negative** – *remote-process* may have crashed.

Notice that only in the positive case is the failure detector guaranteed to be accurate a negative result is correct only probabilistically – there may be a false negative. However, this information is better than no information at all. A second limitation on the information returned by the failure detector is that it may be invalid by the time it is returned to the caller. In particular, a remote process may crash after a failure detector has returned a positive. This limitation does not prevent practical application of failure detectors.

Failure detectors typically are implemented using timeouts. A simple failure detector sends a message to *remote-process*, sets a timer to expire at some point in the future, and waits. If *remote-process* has not failed, it will send a response to the message. If the failure detector receives a response message before the timer expires, it returns positive. If the timer expires before it receives a response, it returns negative (it may receive the response after the timeout, in which case it silently discards it – this is the case where the failure detector has returned false negative result).

Failure detectors operate on an assumption that the upper bound on message transmission is less than half the failure detector timeout. This holds only probabilistically in asynchronous systems. Failure detectors are useful because they expose this probabilistic nature only in the case of negative results. The mechanisms developed in this thesis make use of failure detectors.

2.1.1.6.2 Assumptions about Communication Failures

This thesis takes a high-level view of a link. In this view, links are provided by network transport protocols, which are capable of overcoming omission and Byzantine (lower-level) link failures. The manner in which this is done is described in Section 2.1.3. Higher level communications protocols are then built on top of the transport layer links. The consequence of this view is that for the remainder of the thesis (excluding Section 2.1.3), links can be considered as subject to crash failures only.

2.1.1.6.3 Assumptions about Process Failures

This thesis is interested only in processes that fail by crashing. This failure models such events as power failures and crashes due to programmer errors. Other failure models are useful when a distributed system is executing some particular algorithm, in this thesis, the aim is to provide support for programming distributed systems. In such an environment, programmers using this support are responsible for the algorithms used in the system.

2.1.1.6.4 Assumptions about Network Topology

The mechanisms developed in this thesis are intended to support the construction of systems that operate over internets which are outside the control of the organisation developing the system. Consequently, no assumptions about communication topology are appropriate: the mechanisms described operate over both point-to-point and broadcast networks (and a mixture of the two). In respect of network partitions, it assumed that any loss of connectivity between processes does not persist indefinitely, that is, that partitions, if they occur at all, are only transient.

2.1.1.6.5 Assumptions about Determinism of Processes

The position taken on process determinism is as follows:

- Certain common low-level networking technologies are non-deterministic in some degree (the most obvious example is Ethernet/802.3 [DEC *et.al.* 1980]). Ethernet (in particular) is so widespread that it would unacceptable to prohibit its use in this thesis since the desire is to build general-purpose communication mechanisms.
- However, the Ethernet frame transmission process is contained within the high-level (transport protocol level) view of network links previously introduced. Given that such links place no upper bounds on message transmission delays (ie. the system is asynchronous), the non-deterministic nature of Ethernet (and anything else below the transport level) is subsumed by asynchrony of the network link. Consequently, non-determinism below the transport level can be discounted.
- The transport layer itself is non-deterministic only in such a way as to preclude the ability to place an upper bound on message transmission delay. This has already been precluded in lower layers. Consequently, non-determinism in the transport layer can also be discounted.
- The same is true for non-determinism in the communication mechanisms built on top of the transport layer.
- Finally, application programmers using the communication mechanisms in this thesis might employ non-deterministic processes in their programs[4]. This leads to a non-deterministic system. However, this non-determinism is entirely the responsibility of the programmer who introduced it, and is therefore irrelevant to this thesis.

2.1.2 The Organisation of Network Protocols

Network protocols typically are organised into stacks consisting of one or more protocols at each level (or layer) of that stack. This subsection explains some basic terminology describing the function of the layers in a typical stack.

In general, a given protocol in a given level makes use of services provided by a protocol (or protocols) in the level immediately below. Hence, higher-level protocols provide higher-level services. These services are built on the services provided by lower level protocols.

[4] A fundamental result established in [Fischer, Lynch and Paterson 1985] is that no deterministic algorithm exists to achieve *consensus* (a fundamental operation in fault-tolerant distributed computing) in an asynchronous distributed system which tolerates the crash failure of even one of the processes attempting to reach consensus. A programmer needing a consensus to be established might therefore employ non-determinism (alternatively, failure detectors can also be used).

The *de jure* standard[5] network protocol stack is the International Organization for Standardization's Open Systems Interconnection (OSI) reference model [ISO 1981]. The layers in this model are as follows:

1. **Physical Layer** – concerned with physical transmission of bits over a hardware communication channel.
2. **Data Link Layer** – concerned with the logical transmission of data between computers that are directly connected. Deals with issues such as data loss, corruption and duplication.
3. **Network Layer** – concerned with the logical transmission of data between computers which are not necessarily directly connected. It deals with routing data that needs to traverse several hops in order to reach its destination.
4. **Transport Layer** – concerned with the transmission of data between processes located on computers which are not necessarily directly connected. There might be several processes on a given computer, the transport layer is responsible for getting a given message to the process intended to receive it. The layer is also often responsible for providing point-to-point connexions that reliably deliver a stream of messages to a destination in the order in which they were sent.
5. **Session[6] Layer** – provides a abstraction called a session for (possibly bidirectional) communication between processes. A session is everything a connexion is. It may, or may not, provide extra facilities (such as synchronisation between the communicating processes). If no such extra facilities are provided, and the underlying transport layer supports connexions, the session layer may be virtually non-existent.
6. **Presentation Layer** – concerned with encoding and decoding data transmitted across a heterogeneous network.
7. **Application Layer** – provides protocols supporting programs used by end-users. Such programs include E-mail, file transfer and so on.

The transport layer is often said to be the lowest-level *end-to-end* layer, that is, the lowest level layer which provides a service between ends defined by two processes in a distributed system. Higher level mechanisms, such as those described in this thesis, are most often built upon this layer.

2.1.3 Fundamental Limitations of Transport Protocols

The role of transport protocols is to send information from one process in a distributed system to another. Unfortunately, the potential for failures as described above adversely impacts the reliability of transport protocols, and these limitations impact on the semantics of the higher level protocols using a given transport protocol.

An ideal transport protocol is identified by Lampson [Lampson 1993]. This has the following properties:

- Between any two nodes, messages are delivered in the order sent,
- Messages are always delivered exactly once.
- The sender of a message always receives an acknowledgment of the successful delivery[7] of that message.

[5] The OSI model is not widely implemented, TCP/IP is the *de facto* standard. Nevertheless, the OSI model is appropriate to illustrate layered protocol organisation.

[6] The OSI choice of the term *session* here is rather unfortunate. Outside the OSI model, the term is widely used more or less interchangeably with *connexion*. This thesis will follow this second practice.

[7] Note that message delivery is an action performed by the receiver. This action consists of reading a message off the network, performing any required processing (verification etc) then making that message available to its destination process.

The majority of transport protocols achieve only the first property. This is because, in a realistic failure model where processes can crash, it is not possible to achieve the second and third properties without updating non-volatile state each time a message is received.

To understand why the non-volatile state update is necessary, consider the case where a process, *sender*, sends a message, *mesg*, to another process, *receiver*, and the process *receiver* crashes some time before sending an acknowledgment of *mesg* back to *sender*. At *sender*, there is no way of knowing whether *receiver*, prior to its crash, in fact delivered *mesg*[8]. The only choice is to re-send *mesg*. Now imagine that after *receiver* crashed, another process implementing that same function, called *receiver'*, starts up and receives messages directed at *receiver*. The process *receiver'* will therefore receive the (re-sent) message *mesg*. Should *receiver'* deliver *mesg*? This can only be answered if there is a record of messages delivered by *receiver* in a form that survives its subsequent crash – this record must therefore be held in non-volatile state.

Updating stable state is expensive, and is therefore avoided by most transport protocols. Even if one is willing to incur the expense of a stable state update for each message in order to support exactly-once delivery, it is not clear that the transport protocol is the best place to implement it. The process *receiver* in the above example could log the delivery of *mesg* in non-volatile state and separately from that logging, update some non-volatile state that is specific to the application[9]. What happens if *receiver* crashes after logging the message delivery and before updating state at the application level? One is left with *mesg* delivered at the transport level but without the results of that delivery recorded at the application level. This inconsistency leads to the same kind of trouble as not knowing whether a message has previously been delivered. The solution is to group the application level updates and the delivery logging into a transaction such that either it all is recorded or none of it is recorded. This can only be done at the application level. This thesis leaves the discussion of exactly-once message transport at this point, noting that the most effective way to provide transaction semantics is through the use of special supervisor processes (TP-monitors) which ensure that facilities placed under their charge (such as message transport facilities) achieve the desired transactional semantics. TP monitors provide additional advantages, for example, transactions that span multiple processes, but this is beyond the scope of this thesis.

Returning to the transport level, eliminating exactly-once protocols leaves a choice to provide one of two weaker guarantees. These are:

- *at-least-once* message delivery, and
- *at-most-once* message delivery.

At-least-once delivery works by having the sender periodically re-transmit the message until an acknowledgment is received. Consequently, the target may receive a given message more than once.

At-most-once protocols operate within the context of sessions (or connexions). For a process to send[10] messages to another process, a new session must be established. This involves making a session identifier known to each process. All messages sent within a session are tagged with this session identifier. Sessions are terminated in one of two ways, as described below:

- The two processes agree to the orderly shutdown of the session. In this case, it is certain that all messages in the session were delivered exactly once.

8 If links can fail as well as processes, as previously assumed, one could not even be sure that the receiver had in fact crashed. This is an additional complication which does not effect the need for a non-volatile record of message delivery.

9 For example, if the application is a database management system, some database log records would be written to non-volatile state.

10 As an optimisation, sessions are usually full duplex, allowing transmission in both directions. This is usually cheaper than establishing two independent, simplex, sessions.

- One of the processes in the session does not receive any messages from the other within a given time (either data carrying messages or messages purely for acknowledgment). This may mean that the other process has crashed and message(s) have not been delivered, it may also mean that all messages have been delivered but for some reason the corresponding acknowledgments have not been received by the sender. There is a fundamental uncertainty about the status of unacknowledged messages belonging to a session terminated by failure. If a stop-and-wait[11] strategy is used for message transmission, it is possible to reduce the number of such messages to one per broken session. This method of detecting a broken session is effectively a failure detector, a concept discussed in the previous section.

The important point is that in either of these cases the session is terminated. If a message is ever received containing the session identifiers of a terminated session then that message is discarded. Thus, a process which is restarted after a crash creates a new session (for each process it communicates with) and discards any messages it receives belonging to terminated sessions. In order that this mechanism function correctly, it is necessary that session identifiers be unique. Details of this and other aspects of connexion/session based protocols are given by Lampson [Lampson 1993].

As discussed previously, this thesis is based on the view that transport layer protocols are responsible for overcoming omission and Byzantine link failures and consequently such failures are not exposed to higher-level mechanisms using the transport. The failures are overcome as follows:

- **Failures of omission and re-ordering and duplication of messages** – messages are stamped with a sequence number on transmission. The sequence number conceptually starts at zero and increases by one for each message transmitted. The receiver of a group of out-of-sequence messages can re-establish the proper order using the sequence numbers. The receiver can also request retransmissions of any messages corresponding to sequence numbers that are not found. Finally, messages duplicated by the network link will have the same sequence number, the extra copies can therefore be deleted.
- **Accidental corruption of messages** – the content of each message, including the sequence numbers, is subjected to some hash function. The result of the hash function is transmitted along with the message. The receiver applies the same hash function to the message contents it receives and compares this to the copy of the hash function result it received along with the message. If the two hash results match, the message is considered free of corruption. The probability that this is actually true can be made arbitrarily close to one by expanding the size of the hash function result.
- **Malicious corruption of messages and generation of forged messages** – a link subject to malicious failure can tamper with the contents of messages and then regenerate the hash function result based on the contents after the tampering. Similarly, messages can be forged by the link, which can then apply the hash function. Such messages will pass the test for validity at the receiver. To solve this problem, messages can be cryptographically signed by the process transmitting them, using information (a key) which is not available to the link. Such cryptographic signatures cannot be forged by a malicious link. The receiver of a message will apply a cryptographic function to verify the message in a similar way to the hash function verification. This function might use the same key as the transmitting process used[12]. Alternatively, it might be the public-key corresponding to the private-key used by the

[11] In a stop-and-wait message transmission strategy, message $n + 1$ is not transmitted until after an acknowledgment for message n has been received. This ensures that at most one message is lost in a given session. The drawback is that the effective bandwidth of the link is dependent on the latency, and thus is greatly reduced for high latency links.

[12] This is a shared-key cryptosystem, in this context the shared-key is often called the session key.

transmitting process[13]. If this mechanism is used, malicious corruption and forgery by the link is computationally infeasible. The drawback, however, is that message transmission is significantly slower than for systems where messages are not cryptographically signed. Consequently, signing should be used only in environments where malicious link failures are anticipated.

Consequently, it is reasonable to view transport layer protocols as providing links that are subject only to crash failures.

In summary, at-most-once transport layer protocols are used to support the higher level mechanism discussed in this thesis. These behave as network links that are subject only to crash failures. Furthermore, they provide an integrated failure detector that can be used to detect crash failures either of remote processes or of the links to those remote processes.

2.2 Kinds of Mechanisms

There are several kinds of communication mechanisms within the message-passing paradigm, including the following:

- Remote operations.
- Message queueing facilities (message oriented middleware).
- Bulk data transfer facilities.
- Facilities for the transport of continuous media.
- Facilities supporting one-to-many communication.

Each of these is discussed below.

2.2.1 Remote Operations

Remote operations are the basic building block used in the majority of distributed systems. Two processes are involved, each taking on distinct roles in the interaction: one process is referred to as the client, the other process is referred to as the server. The client sends a request message to the server, this message identifies the operation to be executed and the values of any parameters that are to be passed to this operation. The server receives the message and executes the operation specified in it. On completion of the execution the server sends a (possibly null) reply message back to the client, this contains the result(s) of the operation executed, if there are any.

One should note that the roles of client and server are defined only with respect to a given remote operation. Nothing prevents a process being both a client and a server for different remote operations.

Several other terms are sometimes used in describing this form of communication, including the following:

- Request/reply communication.
- Request/response communication.
- Client/server communication.
- Remote invocation.
- Remote execution.

These terms are essentially interchangeable and will be used as such in this thesis.

2.2.1.1 Approaches to Implementation of Remote Operations

There are two main approaches to the implementation of communication based on remote operations.

[13] This is a public-key cryptosystem. Public key cryptography is several orders of magnitude more computationally expensive than is shared key cryptography. Consequently, the most common approach is to use public-key cryptography to securely exchange a session key and then employ shared-key cryptography with this key for the rest of the session.

- ***Automatically Generated Remote Operations*** – in which remote operations protocols are automatically generated from declarations of the remote sub-programs. The generated protocol typically includes local sub-programs with the same signatures as the remote operations that may be called by client programs to effect remote operations.
- ***Ad hoc Remote Operations*** – in which the developer implements remote operations protocols directly over lower level transport protocols such as TCP.

These are merely the two extremes, it is also possible to use a hybrid approach. For example, a system might employ automation for the majority of remote operations, but use the *ad hoc* approach for a small set of performance critical operations.

Remote procedure call (RPC) [Birrell and Nelson 1984] is the term for a broad class of systems that implement the automated approach. Many RPC systems have been developed, including:

- ***Courier*** [Xerox 1981] – the original RPC system developed by Xerox.
- ***Sun RPC*** [Sun 1985] – used to implement the NFS distributed file system [Walsh *et.al.* 1985] and other distributed systems technologies on Unix-based local area networks.
- ***DCE RPC***– the remote procedure call component of the Open Software Foundation's distributed computing environment (DCE) [OSF 1992].

In the majority of RPC systems, the programmer begins by specifying the services offered by RPC servers. Services are specified in a specialised programming language called an *interface definition language* (IDL). A specification of a service describes the operations that comprise the service. This description is in terms of the operation name, the types, number and mode (input value, output result or both) of parameters, and the type of the result if the operation is a function. The IDL specification may also include other information about the operations or about the service as a whole, for example type definitions.

From the IDL specification of a service, the RPC system generates code for two stubs for each operation. This code is generated in the implementation language used for each system element, this language may be different for different elements. The client stub is a local sub-program to be called by the client in order to effect a remote operation execution. The client stub takes the same number of parameters as the remote operation. Furthermore, these parameters have the same type and mode, and must be passed in the order declared[14]. The client stub takes care of all of the tasks described below:

1. Transformation of any byte-order sensitive parameters, such as integers, from local byte order into network byte order.
2. Serialisation (flattening or linearisation) of any parameters that are complex data structures, such as linked lists, into linear form to enable transmission on the network. This may be quite complex, if the types of the data structures sent admit the possibility of sharing and cycles.
3. Transmission of the transformed and flattened parameters over the network, along with an identifier of the remote operation to be invoked.
4. Reception of the results of the remote operation from the network, in network byte order, and in flattened form.
5. Deserialisation of results.
6. Transformation from network byte order to local byte order.

Steps 1 and 2, and steps 5 and 6 are called marshalling and unmarshalling respectively. Relieving the developer of the need to explicitly write code for marshalling and unmarshalling is one of the great benefits of the automated approach. The other major benefit is that calls to the client stub are subject

[14] Order is important if parameter association is position-based, as it is in almost all programming languages. If a language supports name-based parameter association, as does Ada [ISO 1995], then the order can of course be varied.

to type checking. The type checking rules are those enforced by the language chosen to implement the client. Hence, there is effective type checking of remote operations, assuming that the client's type system is as least as strong as that of the language used to implement the server.

The RPC compiler also generates a server stub (sometimes called a skeleton), this takes care of receiving and unmarshalling parameters, calling the (user-supplied) sub-program implementing the operation, and of marshalling and transmitting results.

One further detail is necessary: the means by which the client locates the server and establishes a data transport to and from it. This may be performed externally to the client stubs, in which case an additional parameter, a server handle, is passed to the client stubs and used for the transmission and reception operations. This has the advantage that the server handle may be reused for several remote operations to a given server. Alternatively, the client stub may itself locate an appropriate server and generate a handle solely for a single, internal, use.

Servers are located by submitted a query to a distributed database generally called a *name service* [Birell *et.al.* 1982, Lampson 1986]. The query specifies a description of the service required (often just a unique identifier generated by the RPC system), and possibly other information such as quality of service constraints. For location dependent services, the server location might also be specified. The name service returns a server handle satisfying the constraints of the query. Subsequent queries with the exact same options might return a handle to a different server, hence the precise server used by a client is abstracted away. Consequently, this approach is called the *service-based* approach, to indicate this abstraction away from the actual identity of the server process.

Abstracting over the actual server used to provide a service has some advantages. If there are several servers implementing a given service, there is a high probability that at least one will be available at any given time. Service-based approaches thus provide fault tolerance. Secondly, the abstraction affords the system the opportunity to select the least loaded of the set of servers implementing a service, hence delivering reduced response time to clients.

In contrast to the RPC/service-based approach, the *ad hoc* approach has little to recommend it. In theory, it can deliver optimal performance because it has direct access to underlying transport protocols. In practice, as is usually the case when abstraction is sacrificed for access to low level details and the possibilities for micro-level performance tuning, the development costs associated with managing these low level details usually outweigh any performance gains. Extra time must be spent dealing with these details, time which is usually more usefully spent at a higher level, implementing macro-level performance tuning. It is possible to develop abstractions to support the *ad hoc* approach [Schmidt, Harrison and Al-Shaer 1995], but the leverage provided is still substantially smaller than that of higher level mechanisms. One particularly significant drawback of the *ad hoc* approach is that it has no built in support for type checking.

Feedback on Reliability

As mentioned previously, exactly-once message transport is expensive to achieve, and, consequently, transport protocols for remote operations usually provide either at-least-once or at-most-once message delivery.

At-least-once behaviour is acceptable only in situations where remote operations which are idempotent. Examples of systems in which remote operations have this property include distributed file systems such as NFS [Walsh *et.al.* 1985]. In such systems, failures are abstracted over and, in effect, exactly-once delivery is simulated. Another way of viewing at-least-once remote operations mechanisms is to consider them as exactly-once mechanisms in which the computations that may be performed in remote operations are restricted to those which are idempotent. Systems in which all operations are idempotent are quite rare, even the otherwise idempotent collection of operations in the original NFS has an associated set of (non-idempotent) operations to provide file locking. Consequently, the more common choice is at-most-once delivery.

It is not possible for a remote operation mechanism built on top of an at-most-once transport to hide failures completely. One approach is for the remote operation mechanism to provide "best-effort" or "maybe" semantics: any operation that is issued has a high probability of being delivered and then of executing. This may be satisfactory for some applications, however it is generally desirable that more information be available at the program level. Extra feedback can be provided in various circumstances as listed below:

- When an acknowledgment for an operation request is received, it is certain that the operation has executed correctly. In this case, positive feedback that the operation has been completed successfully can be made available at the application program level.
- In some circumstances it may be possible to determine with certainty that a request cannot be delivered, for example if the machine on which the target of a remote operation is gracefully shutdown for an upgrade. In this case, it may be desirable to provide program level feedback indicating this fact and thus allow the process issuing the undeliverable request to attempt to execute the request on an alternative server.
- When a session is abnormally terminated, feedback should be provided which indicates those requests that *may* not have been delivered.

There are a variety of ways in which this feedback information might be realised at the programming language level. For example, positive feedback of operation success might be provided by the arrival of the result(s) of a remote operation at the process issuing the operation and negative feedback might be provided by an exception mechanism.

One should also note that even if it is acceptable to incur the expense of providing exactly-once semantics, feedback is still useful. Positive feedback of success indicates not only the successful execution of a request, but also that the request has completed execution[15]. An exactly-once protocol that lacks this positive feedback does not facilitate synchronisation of the client issuing a request with the completion of that request. The negative feedback channel is also useful in an exactly-once mechanism; not to signal transport protocol failures, which of course cannot occur, but rather to signal abnormal termination of the execution of the remote operation. A concrete example of this would be propagation of exceptions raised (and not handled) in the remote operation back to the issuing client. From this it can be seen that the remote operation mechanism built on top of an exactly-once transport protocol should provide very similar language level facilities to one built on top of an at-most-once transport protocol.

2.2.2 Message Queues

One of the drawbacks of client/server remote operations mechanisms in some situations is that in order to perform a remote operation, both client and server must be running simultaneously and network connectivity between the two must be available. An alternative to this approach is to use a continuously available module, called middleware, to act as an intermediary between the client and server.

The earliest examples of message-oriented middleware are message queues. A client wishing to make a request sends that request to the message queue associated with the server for the request. The server for that request need not be running at that time. When the server starts running, it checks its message queue for requests and processes those that it finds. It puts the result of those requests onto a message queue associated with the client(s) that made them. Note that these clients do not need to be running at this time. When a client starts running again, it checks its message queue, and processes

[15] More generally, reception of the reply message by the client indicates that the execution of the request has reached the point where the reply is sent. This is usually the end of that execution, but some systems may allow the reply to be generated at some earlier stage during execution.

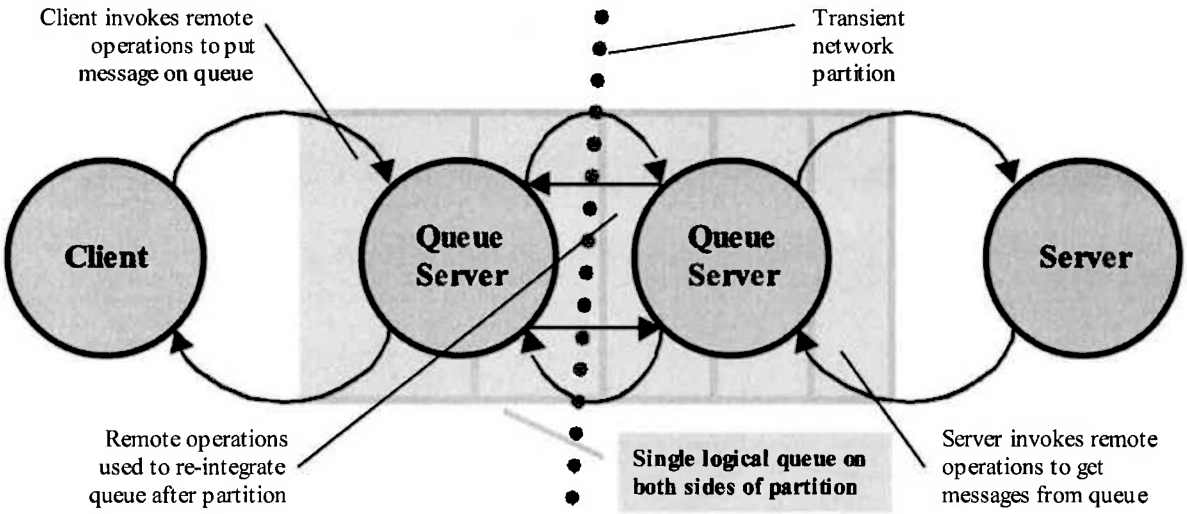

Figure 3 – Implementing a Message Queue with Remote Operations.

any results put there by the server (which, again, need not be running at the time). Message queues therefore remove the need for clients and server to be simultaneously executing, as well as the need for them to synchronise in order to communicate.

At the logical level, a given message queue is a single entity visible to both clients and servers. Physically however, a queue might be distributed. Consider the case where a network partitions periodically so that a client and server lose direct connectivity. The message queue service needs to be available to both on a continuous basis, so it needs to have a presence on both sides of the partition. When the partition abates, the two parts of the message queue automatically reintegrate. In this way, both client and server are provided with the illusion of a single, continuously available queue.

Message queues typically provide an option for some level of stable storage of messages. In and of itself, this may provide increased reliability for communications in many applications. More importantly, stable message queues often provide an standardised resource manager interface. This enables message queue operations to execute under the control of a global TP-monitor. Consequently, message queue exchanges can be part of distributed transactions.

Message queues can be used for one way communication – if message queues provide stable storage, the sender of a request may not need a reply to indicate that the request has been carried out. If the request has been placed into the message queue, and servers remove queued requests only after completely processing them, then the sender is assured that the request will eventually be processed. The sender does not know when it will be processed, but this may not matter for some applications.

Finally, message queues can integrate support for parallelism, fault-tolerant services and load balancing. A given request message queue can be associated with a pool containing several instances of a given server. Each server in the pool can be processing a different message in parallel. If desired, a load balancing function can distribute the work required for processing the messages across the pool. Finally, if some members of the pool stop running, the others can continue to process the queue.

Message queues are very useful in distributed application programming. However, message queues are not a fundamental form of communication mechanism. The reason for this is that efficient implementations of message queue facilities can be constructed from remote operations mechanisms, provided that these mechanism are sufficiently flexible to permit a given process to be both a client and a server. An outline of such a construction is shown in Figure 3.

2.2.3 Bulk Data Transfer

Bulk data transfer mechanisms are specialised for the efficient transport of very large data sets. A typical application is file transfer. Compression techniques may be employed to reduce the time taken to transmit the data. Bulk data transfer mechanisms typically concentrate on optimising throughput, and do not provide the high level support offered by remote operations protocols.

Many bulk-data transfer applications are simply applications of remote operations, which happen to use operations with very large requests and/or responses. In such cases, the distinction between bulk data transfer mechanisms and remote operations mechanisms exists primarily for historical reasons. Adopting remote operations protocols that are capable of transporting request and reply messages of arbitrary size can eliminate this distinction. Distributed file systems account for the largest proportion of data transfer in the majority of local area networks. Typical systems, such as NFS, are implemented on top of RPC or other remote operations mechanisms, rather than using special bulk data transfer facilities.

In typical remote operations mechanisms, the entire contents of a request are determined before any part of the request is sent on the network, and similarly for replies. In effect, there is a quantisation of the data transferred between processes.

In a typical example, a bulk data transfer operation occurs between a producer and a consumer. The amount of data to be transferred might be very large. So large, in fact, that it cannot be buffered in the producer and/or consumer. However, it may be possible that the producer can alternate between computation to produce some portion of the data and transmission of that portion, followed by computation to produce the next portion. If the consumer can also execute in a similar coroutine-like manner, the bulk data transfer and associated computation can be executed even if it is not possible to buffer the whole of the transferred data. To achieve this with remote operations, the transfer would have to be artificially fragmented into several smaller operations.

Even if complete buffering is possible, it may not be the most efficient strategy. For example, a program to retrieve a remote file and write it to local disk is likely to be more efficient if disk writes are interleaved with network reads. This is because this approach uses the direct memory access capabilities of the two relevant IO controllers in parallel. Indeed, the majority of File Transfer Protocol (FTP) clients work in this way.

FTP is a prime example of both the good and the bad points of typical protocols for bulk data transfer. The good point is that it achieves the efficiencies outlined above. The bad point is, that in addition to using a continuous stream for data transfer, a separate continuous stream is used for control signals. The developers of FTP clients and servers must therefore write code to parse the commands in this stream. A significantly superior approach would be to use a high level remote operations protocol to implement command transport and leave the details of parsing, and so on, to that protocol mechanism. It is certainly possible to develop systems using two separate protocols, for example using TCP for bulk data transfer and CORBA for command transfer. As a superior alternative, the OSF DCE remote procedure call mechanism provides the ability to send "pipe" values in remote procedure calls. Pipe values provide the ability to transport arbitrarily long streams of some data type. Once a pipe has been received as a parameter to an RPC, it may be used as a streaming mechanism for bulk data transfer in one or both directions. In addition to its superior integration, the DCE mechanism is strongly typed. This is an advantage over using an untyped transport protocol byte-stream as a bulk data carrier.

In summary, bulk data transfer operations should be supported within remote operation protocols, in two ways:

- Remote operations requests and replies of arbitrary size should be supported, without significantly degrading performance.
- A streaming type constructor, such as DCE's pipe, should be provided in the interface definition language, to support streaming data transfers which are integrated with the remote operations mechanism.

Remote operations protocol support for pipes or a similar facility is a relatively minor concern in comparison to broader issues of remote operations protocols; this thesis concentrates on the broader issues.

2.2.4 Continuous Media

Some distributed systems applications require the transmission of information that is continuous in nature. Examples include the transmission of voice and of video.

Mechanisms for continuous media communication share the goals of low latency and high throughput with other communication mechanisms. In addition, continuous media transmission needs to exhibit *smoothness*, defined as small variation in latency.

A bulk data streaming mechanism which delivers high average throughput and low average latency but which exhibit lengthy breaks in transmission is unsatisfactory for real-time continuous media applications such as video-conferencing. A mechanism delivering reduced throughput and increased latency but providing a smaller upper bound to the length of any breaks in transmission is to be preferred in this case. Continuous media transmission might be integrated into a remote operations framework in a similar way to the integration of streaming bulk data transfer. For example, continuous media streams might be provided by a version of the pipe type constructor offering enhanced quality of service with respect to smoothness.

Continuous media is very much a specialised area. To maintain the focus on general-purpose remote operations, this area is not dealt with by this thesis.

2.2.5 One-to-many Communication

Replication of data across multiple nodes of a distributed system is becoming an increasingly common strategy. Replication is useful for two main reasons:

- To provide fault tolerance.
- To reduce response times via caching.

Replication techniques are primarily used in system level programs, for example, a distributed shared memory facility.

To support efficient replication, specialised one-to-many communication protocols are needed [Hadzilacos and Toueg 1993]. There are two kinds of one-to-many protocols:

- ***Broadcast*** – messages are sent to all nodes on the network.
- ***Multicast*** – messages are sent to a subset of the nodes.

Replication protocols are usually multicast, however the set of nodes which are the multicast targets is usually constant for a given replication system. Therefore, these protocols may be considered broadcast by excluding from consideration those nodes that are not part of the system.

The most basic broadcast protocol is called reliable broadcast. This has three properties as listed below.

- ***Validity*** – any message broadcast by a correct process is eventually delivered by all correct processes.
- ***Agreement*** – all correct processes eventually deliver any message delivered by a correct process.
- ***Integrity*** – every correct process delivers any message at most once and only then if some process (whether correct or failed) broadcast the message.

These constraints imply that if a process fails during a broadcast, either all correct processes will deliver the message or none of them will.

Other broadcast protocols provide additional guarantees. There are three guarantees with respect to message delivery order.

- ***FIFO Order*** – if a process broadcasts a message *m* and later broadcasts a message *m'*, then no correct process delivers *m'* before *m*.
- ***Causal Order*** – if the broadcast of a message *m* causally precedes the broadcast of a message *m'* then no correct process delivers *m'* unless it has already delivered *m*.

  The notion of causal precedence is adapted from Lamport's "happened before" concept [Lamport 1978]. It is defined on significant events, including both broadcast and deliver events. An event *e* causally precedes an event *e'*, written $e \rightarrow e'$, if and only if:

  - a process executes both events, and executes *e* before *e'*, or
  - *e* is the broadcast of a message and *e'* is a delivery of the same message, or
  - there exists some *h* such that $e \rightarrow h$ and $h \rightarrow e'$.

  Note that it is guaranteed that the causal precedence graph is acyclic.
- ***Total Order*** – if two correct processes deliver messages *m* and *m'*, then the first delivers *m* before *m'* if, and only if, the second delivers *m* before *m'*.

Causal order implies FIFO order, however, total order implies neither FIFO order nor causal order. This leads to the six possible protocols shown in Figure 4.

The specifications of protocols given above provide agreement, integrity, and ordering properties only for *correct* processes. It is also useful to prescribe properties for faulty processes. Failures are said to be *benign* if processes may crash and immediately stop and may fail to receive or send messages. Failures involving arbitrary changes of state and collusion between faulty processes are called *malicious*. Given that failures are benign, agreement, integrity, and ordering can be strengthened by the addition of the notion of *uniformity*. Under (non-uniform) agreement, faulty processes may deliver messages that have not been sent by any correct process. Uniform agreement only allows even faulty processes to deliver messages that were sent by correct processes.

There are uniform versions of many of the properties previously defined, these include the following properties:

- ***Uniform Agreement*** – any message delivered by any process is eventually delivered by all correct processes.
- ***Uniform Integrity*** – any process delivers any message at most once and then only if some process (whether correct or faulty) broadcast the message.
- ***Uniform FIFO Order*** – if a process broadcasts a message *m* and later broadcasts a message *m'*, then no process delivers *m'* before *m*.
- ***Uniform Causal Order*** – if the broadcast of a message *m* causally precedes the broadcast of a message *m'* then no process delivers *m'* unless it has already delivered *m*.
- ***Uniform Total Order*** – if two processes deliver messages *m* and *m'*, then the first delivers *m* before *m'* if and only if second delivers *m* before *m'*.

For each of the uniform properties, there is an associated type of broadcast. Note that there is no uniform validity property. Specifying such a property would mean that faulty processes would not be subject to receive-omission failures, thereby severely restricting the scope of application for the specified broadcast.

Broadcast facilities can be integrated into the framework of remote operations mechanisms. Clients may use broadcast facilities to send a remote operation request to multiple servers. Upon delivering such a request, each server executes it as a conventional remote operation and sends a reply message in the same way as is the case for a (unicast) remote operation reply. The client receives all the replies and determines the one(s) to use according to some, possibly application specific, algorithm. For example, this technique may be used to implement replication via a state-machine approach

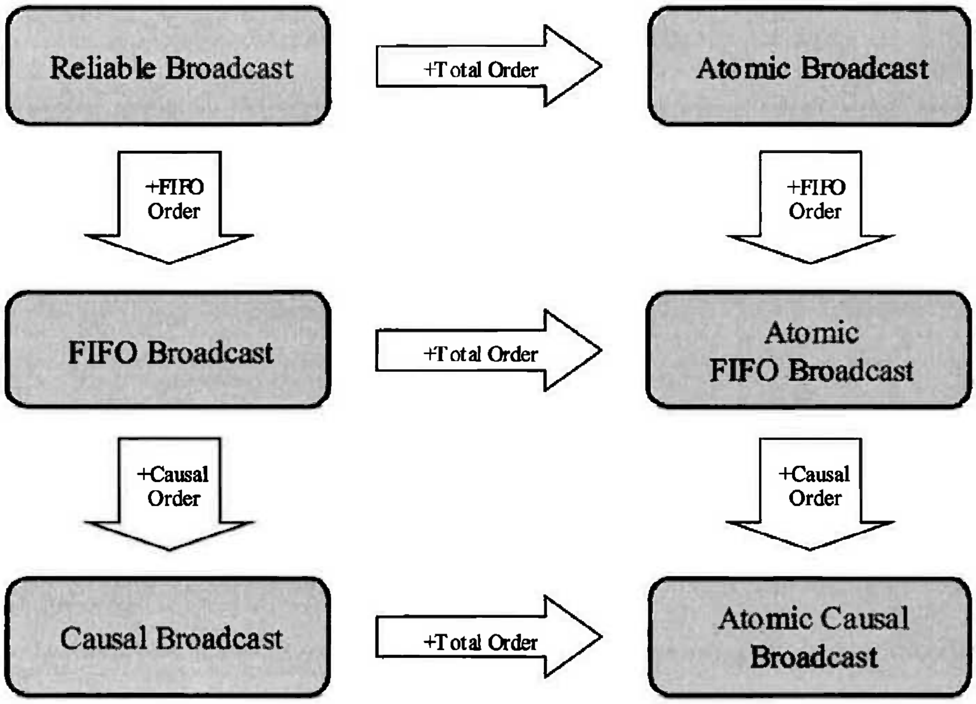

Figure 4 – Broadcast Protocols.

[Schneider 1993]. In this approach, clients broadcast all requests to all replicating servers. Since all servers receive the same requests in the same order, the replies sent by all (correct) servers will be the same. Hence, the correct value of the reply may be taken to be the value of the majority of replies received by the client. At a higher level than remote operations, broadcast communication can be integrated into RPC, with the various broadcast properties being accessible as quality-of-service constraints.

2.2.6 Focus: Flexible Remote Operations

This thesis focuses on the remote operation class of communication mechanisms. As described above, this class of mechanism includes some (non-streaming) bulk data transfer facilities and (given that processes are permitted to be both clients and servers) is in addition sufficient to support message-oriented middleware. Three kinds of mechanisms are outside the focus, these are:

- Streaming bulk data transport.
- Streaming continuous media transport.
- Many-to-one communication.

A strategy to incorporate each of these within a remote operations framework has been suggested above. In addition, each of these technologies serves only a small set of application domains, whereas remote operations are suitable for the majority of distributed systems. Therefore, the focus is neither inconsistent with the other technologies nor overly narrow in application.

2.3 Software Engineering Considerations

Programming languages are some of the most important tools of the software engineer. Although many aspects of software engineering are independent of the particular programming language used, it is clear that modern programming languages, such as Ada [ISO 1995], Java [Arnold and Gosling 1997] and C++ [Stroustrup 1991], provide better support for some aspects of software engineering than do arcane languages such as Fortran. This is because these modern programming languages are designed to support principles of software engineering. Consequently, any new language level mechanism should address software-engineering concerns.

2.3.1.1 Software Quality

Software engineering is a computer science exercise, but it is also a commercial exercise. From the computer science perspective, the task is simply to produce a system exhibiting the highest *software quality*. Commercial constraints, most importantly the time taken to deliver the system and the cost of delivery, militate against the achievement of software quality.

In an idealised software engineering project, the *only* goal is to maximise software quality. McCall *et.al.* consider software quality to consist of several factors, such as those listed below[16]:

- ***Correctness*** – the extent to which a system satisfies its specification and fulfils the customer's requirements.
- ***Robustness*** – the extent to which a system can be expected to continue provide (possibly reduced) functionality under abnormal circumstances.
- ***Efficiency*** – the amount of computing resources required by a system to perform its function.
- ***Integrity*** – the effectiveness of the system in preventing access to system software or data by unauthorised persons or means.
- ***Ease of Use*** – including both the initial effort required to learn to use the system and ongoing effort required to operate the system.
- ***Ease of Defect Removal*** – the effort required in identifying and removing defects in the system.
- ***Extensibility*** – the effort required to add new functionality to the system.
- ***Testability*** – the effort required to develop and deploy mechanisms for testing the system to measure attainment of other quality factors.
- ***Portability*** – the effort required to transfer to the system to execute in a different hardware and/or operating system environment.
- ***Ease of Reuse*** – the degree to which components of the system can be used in the development of subsequent systems.
- ***Interoperability*** – the degree to which the system can be coupled to other systems, in order to exchange data and services.

Abstract software quality factors are not necessarily easy to measure. In fact for some factors, measurement may be impossible. McCall's approach is to identify several *concrete software quality factors*, or *software quality metrics*, which contribute to the abstract software quality factors. These metrics aim to be less subjective than the abstract factors. Some metrics include:

- ***Auditability*** – the ease of verifying the implementation's conformance to requirements.
- ***Completeness*** – the degree to which the system implements the functionality required.
- ***Complexity*** – the complexity of the system design.
- ***Conciseness*** – the amount of source code expressing the system implementation.
- ***Consistency*** – the use of uniform design and documentation techniques throughout the system.
- ***Data Commonality*** – the use of standard data structures and types.
- ***Error Tolerance*** – the effectiveness of the system in managing errors.

[16] McCall [McCall, Richards and Walters 1977] identifies a similar list with very similar factors and broadly similar definitions.

- ***Execution Efficiency*** – the processor time, memory and other resources required in the execution of the system.
- ***Expandability*** – the ease with which system architecture, data structures and operations can be extended.
- ***Generality*** – the breadth of potential application of program components.
- ***Hardware Independence*** – the degree to which the system is decoupled from hardware.
- ***Instrumentation*** – the degree to which the system automatically detects and reports erroneous conditions.
- ***Modularity*** – the degree to which program components exhibit functional independence.
- ***Operability*** – the ease with which the system can be operated by a trained user.
- ***Security*** – the effectiveness of mechanisms to control access to system code and data.
- ***Self-documentation*** – the degree to which source code is, in and of itself, meaningful documentation.
- ***Ease of Understanding*** – the ease with which a new developer can understand the code of the system.
- ***Software Environment Independence*** – the degree to which the system is independent of special operating system and other environmental facilities.
- ***Traceability*** – the ability to trace an implementation component back to a design component and further back to requirements.
- ***Training*** – the ease with which a user can be trained in the operation of the system.
- ***Use of Standards*** – the degree to which the system adheres to standards, and uses external implementations of facilities conforming to those standards.
- ***Ease of Support*** – the degree to which the system requires technical support and the ease with which that support is able be provided.

The relationships between software quality metrics and (abstract) software quality factors are shown in Table 2. Note, in particular, the high degree of influence of the modularity metric. This is expected – the desirability of modularity is probably the most important lesson learned in the field of software engineering to date. Drawing from this lesson, the majority of programming languages designed in the last twenty years contain extensive explicit support for modularity.

Software quality factors sometimes conflict with one another. In addition, since real software-engineering projects are subject to commercial pressures, the attainment of the goals may be further compromised by time and resource constraints. Consequently, managers of software projects must determine an appropriate trade-off between level of satisfaction of each of the goals (and possibly different levels for different sub-systems within the project) and commercial constraints.

2.3.1.2 Automatic Program Checking

This subsection develops three principles governing good design of automatic program checking. The principles are:

- Checking should be as strong as is feasible.
- Checking should be static as much as is feasible.
- Any necessary dynamic checking should be clustered towards the beginning of program units as much as is feasible.

Each of these is explained below.

The requirement that automatic checking should be "as strong as is feasible" is a typical programming language design trade-off. In one view (often that of the language designer), more and

Software Quality Factor / Software Quality Metric	Correctness	Robustness	Efficiency	Integrity	Ease of Use	Ease of Defect Removal	Extensibility	Testability	Portability	Ease of Reuse	Interoperability
Auditability				✓				✓			
Completeness	✓										
Complexity		✓					✓	✓			
Conciseness			✓			✓	✓				
Consistency	✓	✓				✓	✓				
Data Commonality							✓				
Error Tolerance		✓									
Execution Efficiency			✓								
Expandability							✓				
Generality							✓		✓	✓	✓
Hardware Independence									✓	✓	
Instrumentation				✓		✓		✓			
Modularity		✓				✓	✓	✓	✓	✓	✓
Operability			✓		✓						
Security				✓							
Self Documentation						✓	✓	✓	✓	✓	
Ease of Understanding						✓	✓	✓		✓	
Software Environment Independence									✓	✓	
Traceability	✓							✓			
Training					✓						
Use of Standards									✓	✓	✓
Ease of Support					✓						

Table 2 – Relationships between Software Quality Factors and Software Quality Metrics.

tighter automatic checking leads to a reduction of defects in programs that pass the checking, thereby reducing development time and increasing quality. In the contrary view (often that of language implementors and users), are a variety of pragmatic concerns. One concern is that complex checking complicates development of language implementations (and thereby delays language availability). In the extreme case, it may not even be possible to implement a given check. A second concern is that it increases the length of the development cycle for language users because of the additional computation for the checks at various stages of this cycle. This second concern is to some extent in contradiction with the claim of the language designer that additional checking reduces development time. The trend of the past decade in software engineering has tended to favour increased checking – the increased computation required by this checking is more than compensated by the contemporaneous increase in hardware performance. Nevertheless, the first concern is certainly still valid, and there remain certain forms of checking, such as automated correctness proofs, which are not yet feasible even with the performance levels provided by modern hardware. Consequently, the language designer needs to exercise considerable care to achieve balance in this trade-off.

The second principle requires that checking be static as much as is feasible. This also is a trade-off. On one side, static checking improves quality by eliminating the possibility of a class of dynamic failures in statically checked modules, and it improves run-time performance in comparison to dynamic checking, which has additional run-time overhead. On the other side, static implementation of checking is usually more complex than an equivalent dynamic implementation, and, in some cases, a static implementation may be infeasible. Furthermore, there are cases where system semantics require dynamic checking: for example type checking between modules in mutually untrusting domains.

The final principle is that any necessary dynamic checking should be clustered towards the beginning of program units as much as is feasible. When and if the execution of a given program unit reaches a state where all dynamic checks that it contains have been passed, it is proven that that execution of the unit is correct with respect to the automatic checking. Consequently, the remainder of the program unit can be written without the need to account for dynamic failure, thereby simplifying code and enhancing readability. An example of this principle is a program block where all dynamic checks are performed on block entry, and the block exited immediately if any check fails. The corollary is that the code within the block executes only if all checks have been passed. In fact, a module for which all checking is performed statically can be regarded as simply an extreme case of the application of this principle: a case where all checks are performed prior to any execution of the module.

These principles are expressed in general terms applicable to holistic programming language design. However, they are equally applicable to the design of programming language communication mechanisms. The application of the principles is noted where appropriate in the remainder of this thesis.

2.3.1.3 Support for Extended Interactions

Remote procedure call mechanisms support software engineering in two main ways:

- Type checking of parameters and results.
- Structuring messages into requests and corresponding replies, in that sending a request entails that the receiver of that request sends back a reply. This is very important, in that if the expected reply does not arrive (after some length of time), the sender of the request can conclude that there is a problem (network or process failure), and initiate corrective action.

However, RPC mechanisms provide this support only for interactions consisting of a single request and a single (possibly null) reply.

To support software engineering effectively, a mechanism for extended interactions should extend both type checking and message structuring facilities alongside the support for increased flexibility in the message exchange patterns that are possible.

2.3.1.4 Support for Three-tier Client/Server Systems

Advanced client/server distributed systems deployed by corporations typically follow a conceptual structure consisting of a layer of clients and *multiple* layers of servers. The most common multi-layer structure is the three-tier model, in which the tiers are:

- Presentation Tier – Clients.
- Application Tier – Application Servers.
- Resource Tier – Resource (Database) Servers.

The essential idea is to keep application function separate from the data on which it operates.

The procedural information contained in corporate applications provides the "business rules" or "corporate logic" which support day-to-day running of a corporation. These "business rules" act on data stored in a variety of corporate resources. The most important kinds of corporate resources are databases, other kinds of resources include E-mail, file services etc. In a large corporation, there are typically a multitude of such resources, some providing information relevant to the corporation as a whole, some serving only a particular division or unit.

Individual application systems typically make use of resources held on more than one server, and must often be accessible from several client platforms, necessitating versions of the client software for those several platforms. A given business rule may be applicable in several individual applications.

The majority of early client/server systems are two tier systems, in which the two tiers are as follows:

- Client Tier - Clients, running on desktop computers.

- Server Tier - Resource servers, running on server computers. A resources server is typically built around a core consisting of an off-the-shelf product, such as a relational database management system from a commercial vendor.

In a two-tier, client/server system, application logic must located within clients, or within resource servers, or spread between clients and resource servers.

If application logic is solely resident in clients, as is most common in two-tier systems, a given business rule must be implemented in each client for each application which is dependent on it. This version of the two-tier model has a number of drawbacks:

- **Increased Cost of Initial Development** – Each client must include sub-systems for interacting with each resource server that manages resources that it uses. Furthermore, because resource servers are typically constructed from the basis of a commercial product, this may involve using a different proprietary protocol for type of each resource server from a different vendor[17]. This leads to increased software development cost.
- **Increased Cost of Extension** – Through its lifetime, the scope of deployment of a given application may be extended. For example to cover the entirety of an enterprise rather than just a division. The client(s) supporting that facility must interact with a increased set of resources, and, possibly, an increased set of resources. In this latter case, the implementation of client(s) must be revised solely to add support for these resources.
- **Increased Cost of Revision** – As a corporation's needs change, so also do the business rules supporting those needs. If a given business rule changes, all client programs using that rule must be first identified and then updated.
- **Increased Cost of Deployment of Revised Systems** – Client software runs on each user's workstation. In a large corporation, there may be tens of thousands of such workstations, each running several different client programs. After a given client program has been revised, the task of deploying the new version of the software to each user's workstation remains non-trivial.

The impact of these drawbacks increases with the number of client seats. For large corporations, enterprise-wide systems using this two-tier approach are likely to prove to be infeasible.

Alternatively, the application logic might entirely be contained in the resource servers relevant to the application. This version of the two-tier model also has a number of drawbacks:

- **Increased Cost of Initial Development** – The location of application logic within resource servers involves the use of server facilities, such as stored procedures in the relational database case, which are in all probability vendor-specific. Consequently, even if the application logic is contained within resource servers, the implementation of clients must involve server-vendor-specific code.
- **Increased Cost of Extension** – Systems with resource server application logic are susceptible in the same way as those with client application logic to increased cost when additional resources are added to the systems. Furthermore, because the application logic resides within the resource servers, a copy of this logic must be added to any resource server prior to integrating it within an existing application system,

[17] This is true even in the relational database context, in which the use of SQL constitutes a supposed standard that is independent of database server vendor. The reason is that each vendor offers proprietary extensions to SQL (and will continue to do so, as such extensions are one of the primary arenas for competition). In order to maximise performance, application developers tend to make heavy use of these extensions, thereby leading to an interface between client and server that is, at the least, partially server-vendor specific.

- **Increased Cost of Revision** – As with client-side application logic, there are several copies of a given business rule, in this distributed across all resource servers holding resources subject to that rule. If a given business rule changes, all resources subject to that rule must be identified and then the server application logic updated.
- **Requirement for Coordination between Resource Servers** – Certain business rules may involve access, and possibly update, to resources held within more than one resource servers. If application logic is contained within clients, then it is generally fairly straightforward for the client to coordinate accesses to these servers. However, if application logic is held within resource servers, then that resource server must be responsible for the coordination. This implies, at the least, that resource servers export a second interface for inter-server coordination, in addition to the regular interface exported to clients[18].
- **Increased Load on Server Computers** – It is often said of typical distributed computing environments that "clients have the cycles to burn"[19]. In contrast, processor cycles on server computers are typically relatively precious, in part because hardware for typical resource servers, such as database server, is specialised for disk input and output, rather than for processing. The location of application logic on resource server computers may therefore not be a particularly wise use of available computing resources.

The impact of these drawbacks increases with the number of resource servers in a corporation. For large corporations, as with client-side application logic approach, enterprise-wide systems using the two-tier approach with application logic located on resource servers are likely to prove to be infeasible.

A hybrid approach, in which application logic is spread over both client and resource servers, may ameliorate the problems to some extent, but does not attack the fundamental cause, which is the duplication of application logic.

In the three-tier approach, the core functionality of a distributed system, incorporating the business rules, is contained within the application server tier. Ideally, a given business rule should be implemented by only a single application server program; if other application servers require access to this business rule, they should gain it by acting as clients for the server containing it. Interaction in a three-tier system is typically supported by system-wide standard mechanisms, commonly called "middleware". The middleware has responsibility for abstracting over the differences between database servers. Since application servers are typically developed in-house, unnecessary differences in the service interfaces they present to clients can be minimised. The three-tier structure employing middleware has the following advantages in comparison to the two-tier structure:

- **Centralised and secure control over business rules** – All business rules are held in central facilities and are therefore easily accessible. Furthermore, these central locations can be secured to prevent unauthorised access and tampering.
- **Decreased Cost of Initial Development** – Clients interact with application servers following a simple protocol which is specialised for the particular application and hence does not incur the overhead of a more general protocol. The middleware mechanism is responsible for generating this protocol, in response to developer declarations. Furthermore, since application servers use middleware to interact with database servers they may abstract over arbitrary differences in those servers and interact with all such servers in the same way. Finally, the application

[18] Note that appointing one resource server per application as the coordinator for that application is a (rather inelegant) version of the three–tier approach, in which the appointed coordinator is an application server as well as a resource server.

[19] The quote is from [Satyanarayanan 1993], in the particular context of distributed file systems, but is applicable to distributed systems in general.

functions implementing a given business rule need be developed only once, as part of one application server program. Other application servers needing that business rule can access it by acting as clients to the application server containing it.

- **Decreased Cost of Extension** – Application servers use middleware to interact with database servers. If a new database is added to a given system, only minor changes are required to the application servers that will use it. In particular, these application servers to not have to be revised to incorporate the details of interaction with the DBMS serving the new database.
- **Decreased Cost of Revision** – If a business rule should change in response to changing requirements, it is only the application server implementing the particular rule that need be revised. Clients need only be revised when extra functionality is added to the system; a much less frequent occurrence.
- **Decreased Cost of Deployment of Revised Systems** – As indicated above, most revisions of a three-tier system affect only the application servers. There are typically an order of magnitude less instances of a given application server than there are instances of a given client program. Furthermore, the hosts for application servers are typically more formally administered than are user's workstations, and therefore it likely to be easier to apply any required revisions. Consequently, deployment of a revised application server is much less difficult than deploying a revised client.
- **Increased Scalability** – Three tier systems employ centralised processing of application logic, however, instead of being performed on resource servers specialised for disk input and output, this processing is performed on server hardware specialised for processing and network input and output. This location of application processing is inherently less expensive, due the appropriate specialisation of the hardware. It is also more scalable, due to the fact that it is usually possible to deploy multiple instance s of a given application server, interacting with a single shared instance of each resource server.
- **Increased Security** – The location of application logic within clients may compromise the security of that application logic, in that a knowledgeable disaffected employee could tamper with the client-side logic. Location of the logic on a centrally controlled server significantly curtails exposure to this risk.

It should be noted that three-tier systems do not necessarily consist of a layer of clients and two layers of servers such that a given layer makes requests only of the layer below it. Application servers may request services from other application servers. Application servers may be able to perform certain functions without making requests to database servers. An additional kind of server, the router, may be interposed between clients and applications servers to avoid clients having to maintain a connexion to each application server they use. Instead, a client connects only to a router, which then manages the forwarding of requests to an appropriate server. The router may also provide other functionality, such as load balancing between a collection of instances of a given application server, and starting and terminating instances in response to variations in demand.

To support the engineering of robust, scalable, distributed application systems that are capable of evolving to support changing business requirements, a communication mechanism must be able to support systems involving multiple layers of server components.

2.3.1.5 Appropriate use of Multiple Threads of Control

The traditional model of a process contains only a single locus of execution, represented by a program counter value, an execution stack pointer, a set of register values and a status (running, blocked, ready-to-run, terminated *etc.*). In addition to this locus of execution, a process contains many other items of execution state: an address space with associated memory projections, references to operating system resources such as open files and system call state. A time sharing operating system will typically support the concurrent execution of tens to hundreds of process, using context switches to divide scarce CPU resource between those processes. Since processes have a significant volume of execution state, the time to perform a context switch is rather large.

In many distributed systems applications, it is advantageous for a node to contain multiple execution loci – the most obvious example is a server that is able to serve several client requests concurrently. A *thread* is an abstraction of a locus of execution: its execution state typically does not include all the items in a process's execution state. There may be several threads within a process. Each thread holds some execution state privately, and the remaining items of execution state are held at the process level and shared between all the threads of the process. Consequently, the amount of context that must be switched to transfer execution between two threads in the same process is significantly reduced, and so context switch times are smaller. Each thread within a process has its own execution stack. The static data of a process is shared (as is executable code!). The heap is also generally shared, although there may be a mechanism to support a private heap for each thread. Threads can be characterised as lightweight computations, whereas processes are heavyweight computations. The precise division of the execution state between threads and processes varies somewhat, however, the address space and operating system resources (such as files) of the containing process are almost always shared[20].

This section describes the circumstances in which multi-threading is consistent with software-engineering goals. More importantly, it identifies some common cases where it is inconsistent. First, however, the characteristics of different kinds of threads mechanisms are examined.

There are several ways to support multiple threads within a process, typically these fall into three categories:

- User-level threads.
- Kernel managed threads.
- Hybrid schemes.

Each of these schemes has advantages and disadvantages, discussed below. In addition to this, the special issue of gang scheduling is discussed. Finally, having identified the characteristics of the various approaches to multi-threading, the issue of how (and how not) to use threads within distributed systems is addressed from a software engineering perspective.

2.3.1.5.1 User-level Threads

Threads may be implemented entirely at the user level (usually in code located in a library). In this case, the kernel knows nothing about the threads within a process, only about the process itself. The consequences of this are listed below.

- Threads are unable to perform system calls independently. If any thread in a given process executes a (blocking) system call, all other threads in that process are also blocked for the duration of the call.
- Similarly, threads are unable to incur page faults independently.
- Multiple threads within a given process cannot be executed in parallel on a (shared memory) multi-processor.

[20] These limitations are a fundamental part of the process notion. Systems which remove the limitation, such as Grasshopper [Rosenberg *et.al.* 1996], generally also remove the notion of process. For example, Grasshopper has two orthogonal abstractions, *locus*, providing an agent of execution, and *container*, providing a address space. Loci can dynamically migrate between containers, and also create new containers. In addition, Grasshopper is designed to support orthogonal persistence [Atkinson *et.al.* 1983], and consequently does not have any concept of files.

- Since the threading mechanism is entirely user-level, programmers with specialised needs can replace any part of it with an equivalent part of their own. In particular, it is reasonably easy to implement a customised scheduling mechanism.
- A thread's private execution state excludes any kernel resident information, hence the thread's state consists only of register values, with the stack pointer referencing a private execution stack.
- Synchronisation between threads is implemented without kernel support, using interlocked instructions such as *atomic swap* and *test-and-set*.

The first three consequences are clearly limiting. Non-blocking system calls can be used to overcome the first limitation, however, this usually leads to a more cumbersome programming style which is more difficult to understand. Little can be done about the blocking effect of page faults, since the page fault handling mechanism is generally within the kernel and therefore inaccessible[21].

In summary, user level threads are mainly suitable for uni-processor computations that do not execute system calls, in other words for compute intensive processes. A typical example is a discrete event simulation, which may incorporate thousands of processes. User-level threads offer particular advantages in the form of very fast context switches and through the ability for a programmer to replace the thread-scheduling algorithm.

2.3.1.5.2 Kernel Managed Threads

Kernel managed threads are somewhat more heavyweight than are user-level threads. Typically each kernel-supported thread has its own system call and page fault state. The kernel implements scheduling and context switching of kernel managed threads. The characteristics of the kernel managed threads approach are listed below.

- Threads may perform blocking system calls and incur page faults independently.
- Threads in the same process may run in parallel on a multi-processor.
- Context switches between threads in the same process require transitions into and out of kernel mode because the scheduler is inside the kernel. Consequently, kernel thread context switches are much slower than context switches between user-level threads in the same process.
- Even ignoring the time taken to switch in and out of kernel mode, the time taken to perform a context switch is somewhat larger than it was for user-level threads because of the increased amount of thread execution state. In general, this is less significant than the cost identified in the previous point.
- Threads within a given process can have different memory protection contexts. This is feasible since context switches are done in kernel mode, it is easy to adjust protection contexts during a context switch. If context switches are done in user mode, the adjustment of protection contexts must be done via (multiple) system calls, leading to an unacceptable increase in context switch time.
- Programmers have limited control over thread scheduling and other aspects of the threading mechanism.

In summary, these characteristics mean that kernel threads generally provide superior performance for I/O-intensive programs and possibly for compute-intensive programs on multi-processors, but inferior performance for compute-intensive programs on uni-processors.

[21] Some modern operating systems, most notably Mach [Acetta *et.al.* 1986], support the concept of a user-level page fault handler. However, such operating systems provide kernel supported threads anyway so the point is moot.

2.3.1.5.3 Hybrid Approaches

Hybrid mechanisms attempt to provide the advantages of both schemes. The simplest hybrid approach implements user-level threading within each kernel thread. In this approach, a process consists of a number of kernel threads, each containing one or more user-level threads. Applications are implemented in terms of user-level threads. Scheduling of the user-level threads within a given kernel thread is entirely under programmer control, and context switching between these threads is very fast. However, user-level threads within a given kernel thread are unable to block independently, although user-level threads in different kernel threads can. Furthermore, user-level threads within the same kernel thread cannot execute in parallel.

An improvement on the previous approach is to eliminate the containment of user-level threads by kernel threads and allow a given user-level thread to execute within any of a number of kernel threads (possibly any available kernel thread). This is basically the approach taken in Solaris[22] [Sun 1997c] and has advantage that any n user-level threads can block independently and execute in parallel on a multi-processor, where n is the number of kernel threads available[23] to execute user-level threads. A user-level scheduler can have complete control over the constitution of the set of user-level threads that is executing at a given time, but not the size of that set[24]. This user-level scheduler can perform fast context switches between user-level threads. However, the number of user-level threads that can be (in effect) blocked in the kernel is dependent on the number of kernel threads allocated by the process. Therefore, in order for any user-level thread to be guaranteed to block independently, the number of user-level and kernel threads must be the same, with any user-level thread able to run on any kernel thread. In this hybrid system, the kernel schedules (virtual) processors represented by kernel threads and the user-level run-time system schedules user-level threads on these virtual processors.

2.3.1.5.4 Gang Scheduling Issues

Consider a system with two processors and four runnable threads, $T_1 \ldots T_4$, where T_1 and T_2 are threads within the same process, P. Each of T_1 and T_2 periodically requires exclusive access to a shared buffer in memory (perhaps they are a producer/consumer pair). Assume the system assigns processors such that one processor is multiplexed between T_1 and T_2 (hence, P obtains 50% of the available CPU). Then any attempt to access the shared buffer whilst it is locked will require a context switch to the other thread and back again before it can be granted. A context switch will also be required if one of the threads has emptied or filled the buffer and is consequently unable to proceed until the other thread has done some work. All these extra context switches mean that the throughput of P is reduced (since time is spent in undertaking context switches rather than real work).

In contrast, if the system schedules processors, so that 50% of the time, both T_1 and T_2 have processors, and in the remaining 50% of the time neither of them do, the additional thread context switches identified above will be avoided. As a result, the throughput of P will be increased, even though P is still receiving exactly the same amount of processor time. In this example, threads T_1 and T_2 constitute what is known as a *gang*, and the technique of scheduling them both at the same time is called *gang scheduling*. In the general case, the number of processors is greater than two and the gang size is less than or equal to the number of processors.

In the above example, the non-gang-scheduled approach gains a performance advantage because it is never necessary to perform a context switch of P. Since process level context switch times are

---

[22] In Solaris, kernel supported threads are called lightweight processes (*LWPs*) and user-level threads are called threads.

[23] In Solaris, it is not necessarily the case that all *LWPs* can execute any thread. A particular set of *LWPs* might be bound to execute only a particular set of threads (most often the size of both of these sets is one).

[24] Allowing this would imply that processor allocation is controlled at the user level, which is not generally considered an acceptable alternative!

	Process	User-level Thread	Kernel Thread	Hybrid Thread
Program Counter	yes	yes	yes	yes
Stack Pointer	yes	yes	yes	yes
Registers	yes	yes	yes	yes
Status	yes	yes	yes	yes
Address Space	yes	no	no	no
Memory Protections	yes	no	some	some
Child Processes	yes	no	no	no
Child Threads	not applicable	yes	yes	yes
Timers	yes	some	no	some
Signal Management	yes	some	no	some
Open Files	yes	no	no	no
System Call State	yes	no	yes	yes
Accounting Information	yes	some	no	some
Static Data	yes	no	no	no
Heap	yes	some	some	some
Stack	yes	yes	yes	yes
User-level Scheduling	hint based	algorithmic	hint based	limited algorithmic + hint based
Kernel Context Switches	yes	no	yes	some
Multi-processor	yes	no	yes	yes
Gang Scheduling	hint based	none	hint based	limited algorithmic + hint based
System Calls Independent	yes	no	yes	yes
Page Faults Independent	yes	no	yes	yes

Table 3 – Comparative Characteristics of Threads Mechanisms.

several times larger than thread context switch times, this may appear to be a flaw in the analysis. In practice, this is not the case, for two reasons:

- The number of extra thread level context switches required by the non-gang-scheduled approach is several orders of magnitude greater than the number of process level switches required by the gang scheduled approaches, and,
- More importantly, the number of runnable processes on typical multi-processors is significantly larger than the number of available processors. Consequently, processes generally do not receive 100% of even one processor, and hence the non gang scheduled approach will not avoid process context switches.

To take advantage of multi-processor techniques the threads mechanism must be multi-processor aware; this eliminates user-level threads from consideration. Both kernel managed threads and conventional processes (presumably communicating in some way) can support gang scheduling. In these cases, the programmer requests gang scheduling by providing information in the form of hints to a kernel based scheduling algorithm. In hybrid multi-threading mechanisms, hints are used to control the allocation of processors to a process, and the scheduling of threads within that process to the available processors can be completely controlled by the programmer. Consequently, the hybrid approach supports the most effective gang scheduling, particularly in complex situations, such as a process containing several independent gangs of threads.

Table 3 shows the advantages and disadvantages of each of the basic user-level and kernel managed alternatives. Hybrid schemes can effectively combine the advantages of each of the basic alternatives, and in some areas, most notably support for parallel computation and gang scheduling, are better than both basic alternatives.

2.3.1.5.5 Software Engineering with Threads: Effective Usage

The use of multiple threads can be an indispensable tool for the software engineer, providing the opportunity for improved performance through concurrency and improved clarity of program structure. Threads can also have a detrimental effect on software quality. This subsection examines several examples in which the use of threads is effective with respect to software engineering goal and

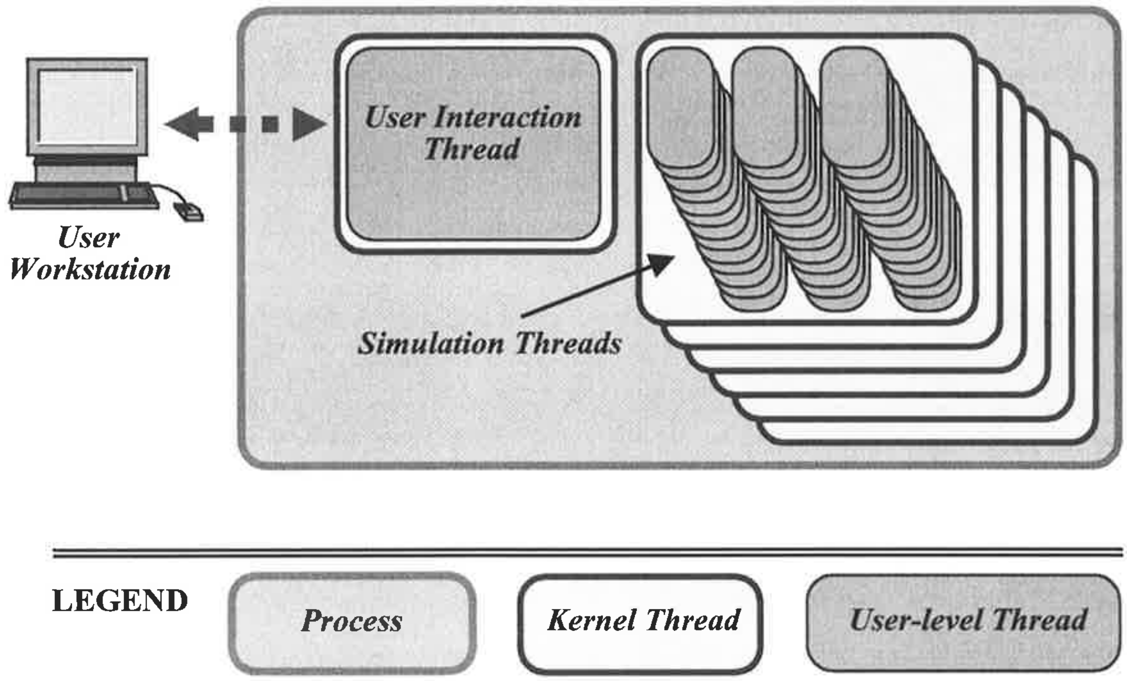

Figure 5 – Typical Thread Structure of a Multi-threaded Discrete Event Simulation.

attempts to identify commonality. Some examples of systems where multi-threading works well are listed below:

- Discrete event simulations containing thousands of compute-only threads.
- Multi-threaded servers in which multiple threads are used to service several client requests concurrently.
- Desktop productivity applications in which threads are used to provide constantly available interactivity whilst time consuming activities are performed in the background.
- Computationally intensive scientific applications in which multiple threads are executed in parallel to solve different partitions of a problem. This is known as *data parallelism*.
- Computationally intensive processing applications, such as image processing or computer vision, in which threads are used to perform different processing functions in parallel in a pipelined fashion. This is known as *functional parallelism*.

Each of these models of multi-threading will be examined in turn.

Discrete Event Simulation

Discrete event simulations are naturally expressed as a collection of computations communicating via an event queue (or queues) [Ashenden, Detmold and McKeen 1994a,1994b]. There may be thousands of these computations in a given discrete event simulation. There is a straightforward mapping from these computations to threads in the implementation domain.

On uni-processor computers, the main performance goal is to minimise the percentage of time spent in context switches between threads and hence maximise use of available processing time and provide minimal simulation run time. This goal is best achieved by a user-level threads mechanism, possibly with the scheduler and other aspects of the threads package specialised for discrete event simulation. On a multi-processor, a hybrid threads mechanism should be used, with user-level threads as before, a number of kernel threads equal to the number of processors and user-level threads able to run on any kernel thread. This design, with an additional thread to handle user interaction, is shown in Figure 5.

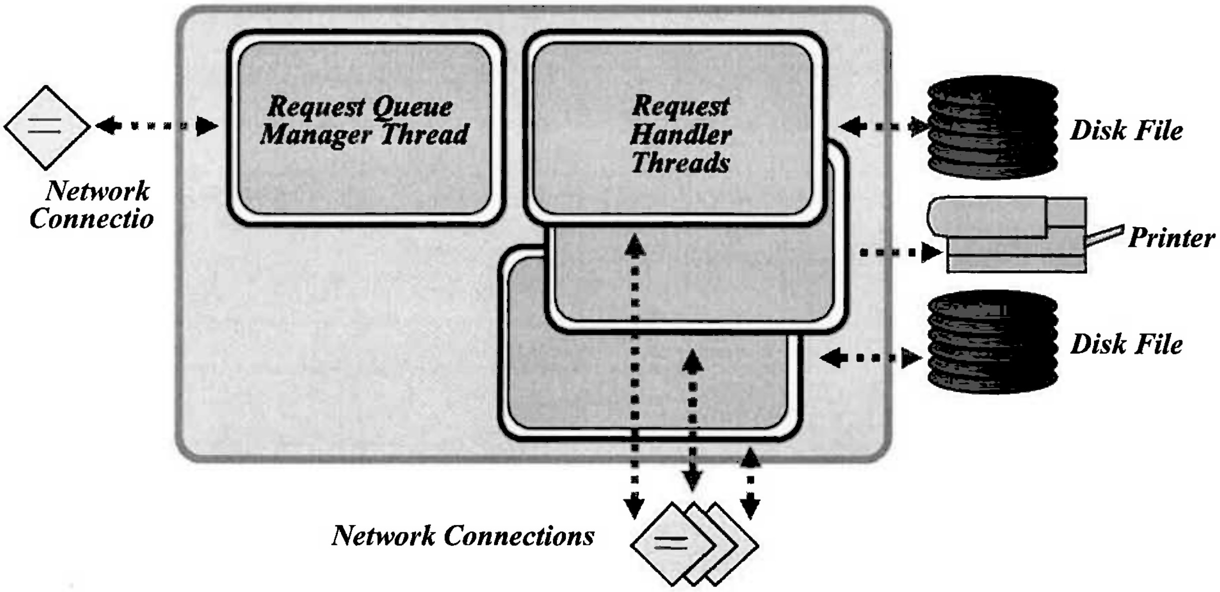

Figure 6 – Typical Thread Structure of a Multi-threaded Server.

Multi-threaded Servers

A server in a distributed system can receive further client requests whilst engaged in the service of a client request. There are three alternative strategies for the development of the server with regard to this situation.

- ***Single-threaded Server*** – The server can ignore the other request(s) until it has completed the service of the current request.
- ***Finite-state Machine Server*** – The server is single threaded. However, when the service of a request requires a system call, a non-blocking system call is made. If this call returns with the operation incomplete, the server switches to undertake some work on another request. The server later switches back to perform further work on the first request.
- ***Multi-threaded Server*** – The server uses a different thread to service each request. The implementation typically uses blocking system calls.

The first of these alternatives is easy to understand but offers poor performance for two reasons:

- Nothing is being done whilst the server is blocked in a system call. This time could more profitably be used to service a subsequent request.
- There is no opportunity to perform parallel computation.

The finite-state machine alternative solves the first of these problems – no time is spent blocked in system calls if there is any work able to be done. If asynchronous I/O facilities such as those defined by the real-time extensions in the IEEE POSIX standard [IEEE 1996] are available, these can be used to perform I/O in parallel with computation. However, this alternative still does not provide an opportunity to perform the computations associated with different requests in parallel. Furthermore, there is a new problem:

- The use of non-blocking and asynchronous I/O and the state-machine structure leads to a system that is significantly more difficult to understand, even at a high level of abstraction, than the single threaded server.

The multi-threaded server design solves all three problems. The use of a kernel thread to service each request means that blocking system calls made in the service of one request do not impede the service

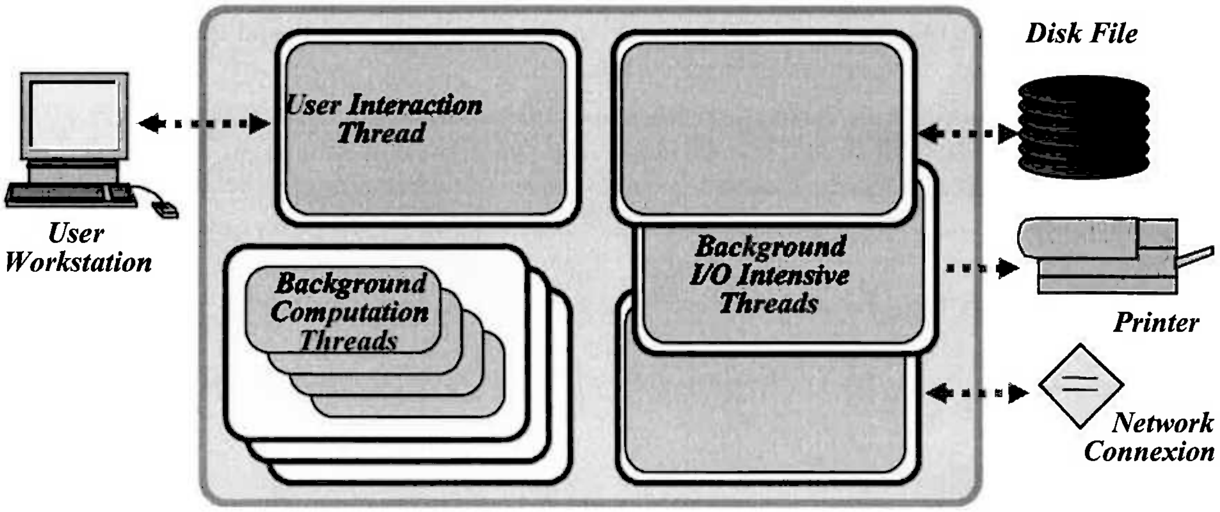

Figure 7 – Typical Thread Structure of an Interactive Client Application.

of another request. These kernel threads can also run in parallel on a multi-processor, permitting parallel computation. Finally, a multi-threaded server is often not significantly more complex than the original single-threaded server. In particular, the code for handling each kind of request is likely to be the same. In some multi-threaded servers, there may be additional complexity due to a need to synchronise concurrent requests. This complexity would be required in an equivalent server based on a finite-state machine as well – it is unavoidable if multiple non-independent requests are to be serviced concurrently.

Figure 6 shows a multi-threaded server using hybrid threads with each user-level thread contained in a different kernel thread. There is a single request queue manager thread responsible for receiving requests and then dispatching these to a request handler thread to be serviced. The request queue manager might create request handler threads on demand, or it might maintain a pool of request handlers, which can be re-used after they have completed a service. On a multi-processor machine, this design enables the computation of several requests to proceed in parallel. Systems structured in this way are sometimes said to exhibit *task parallelism.*

Interactive Applications

Threads can be used to improve interactivity and performance in applications such as desktop productivity tools and network clients. Figure 7 shows an multi-threaded interactive application process. A user-level thread with a dedicated kernel thread provides continuous interactive response. Long running pure computations, such as recalculating a spreadsheet, are handled by a collection of user-level threads. To support parallel computation these threads can execute on any of several kernel threads. Long running tasks requiring I/O are handled by separate user threads, each bound to its own kernel thread. Examples of such tasks include saving a file, printing a hard copy or interacting[25] with a remote server over a network to find and retrieve some data.

Data Parallel Applications

In some application areas, notably scientific and mathematical computing, it is often possible to partition the data of an application into several parts. Computation is then performed on each of these parts independently, before finally recombining the parts to yield a result. When the computation on each part is performed by a separate thread in parallel with other threads performing computation on other parts, the structure is known as *data parallelism.* Figure 8 shows a data parallel application with

[25] This interaction should involve more than simply sending a remote operation request and blocking until the reply arrives. Typically, this interaction will include several remote operations with some intervening (or overlapping) computation.

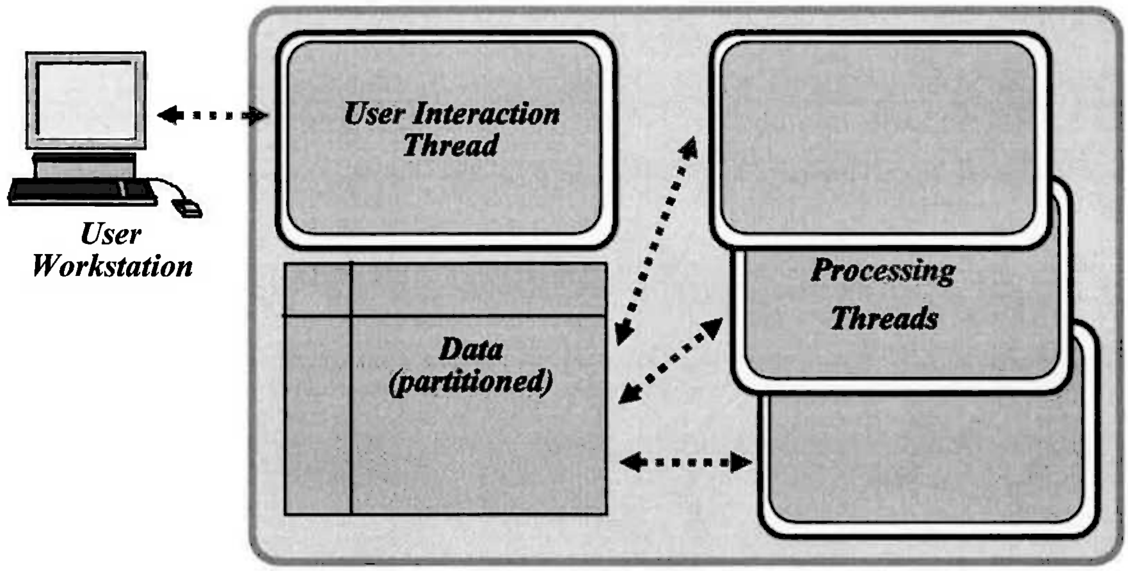

Figure 8 – Typical Thread Structure of a Data Parallel Application.

the data partitioned into three parts, a processing thread for each part and an additional thread managing user interaction.

Functionally Parallel Applications

Another way to partition an data processing application to exploit parallelism is to divide the data processing task into a number of stages arranged into a pipeline. Data flows through the pipeline being processed by successive pipeline stages. Once the pipeline is full, each pipeline stage is processing a different datum. Kernel level threads can be used to make the different stages run in parallel on a multi-processor. This structure is called *functional parallelism.*

Figure 9 shows a functionally parallel application with a three stage processing pipeline and an additional thread managing user interaction as well as acting as both the source and the sink of the data passed through the pipeline. The connexions between source and sink, and adjacent pipeline stages, typically are implemented by shared queues.

Combinations

The three fundamental parallel processing structures identified above (task parallelism, data parallelism and functional parallelism) are able to be combined. Suppose a stage of a pipeline is too slow to keep up with the rate at which data are produced by the previous stage, and, further, that it is not possible to split the stage into several stages. In this case, it may be possible to build a parallel implementation of the stage using task parallelism or data parallelism. Similarly, a task in a task parallel structure may actually be implemented using several data parallel or functionally parallel threads.

Reasons for Effectiveness of Threads in the Examples

There are several reasons for the effectiveness of threads as a tool for implementing the systems described above.

- ***Thread Isomorphism*** – in the discrete event simulation, multi-threaded server and interactive client examples, the threads in the implementation domain correspond to identifiable activities in the analysis and design domains. This property is identified as the *traceability* metric of software quality in Section 2.3.1.1. As seen in Table 2, traceability, (thread isomorphism in this case), contributes both to the correctness and to the testability of the software produced.

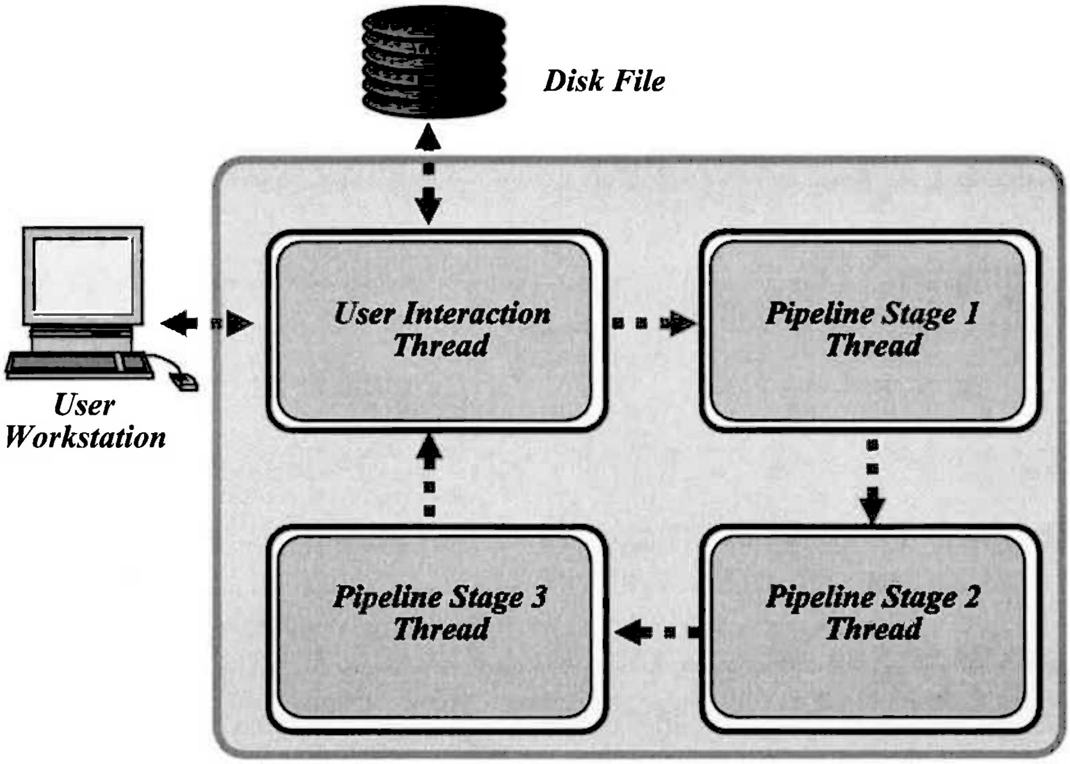

Figure 9 – Functionally Parallel Application with a Three-stage Pipeline.

- ***Appropriate Thread Granularity*** – in each of the examples, the granularity of the computation performed by a thread is such that the overhead of the chosen threads mechanism does not significantly degrade performance.
- ***Improved Interactive Response*** – in the interactive client, a dedicated thread is used to provide satisfactory response time for human interaction. This is an important part of successfully engineering software with a human interface (*cf.* the *operability* software quality metric in Section 2.3.1.1).
- ***Parallel Processing*** – in all of the examples, threads are used to take advantage of multi-processor hardware in order better to meet performance-related goals of software engineering.

In the specific case of distributed systems it has been demonstrated in this chapter that threads have a role in both clients, to improve interactivity, and in servers, to improve throughput.

2.3.1.5.6 Thread Abusage

The discussion above has shown that multi-threading is a valuable software-engineering tool when used appropriately. Unfortunately, threads can also be used inappropriately, as shall be demonstrated.

File system I/O system calls, such as *read()* and *write()* in Unix, often take a relatively long time to complete. A common approach when there is a need for a thread to do an I/O operation asynchronously is to create a child thread solely for that single operation. The child thread performs the I/O operation synchronously and then waits[26]. The parent continues with some computation and then synchronises with the child (and hence with completion of the I/O). The child then exits and the parent continues (using the result of the I/O, if any). The child retrieves certain information from the system call: the data read in the case of a read operation and the knowledge that the write operation

[26] For asynchronous read operations, the parent could give the child a memory buffer and associated locked *mutex*. The child would unlock the *mutex* after the read finished and then exit. The parent would synchronise with the completion of the read by locking the *mutex* a second time.

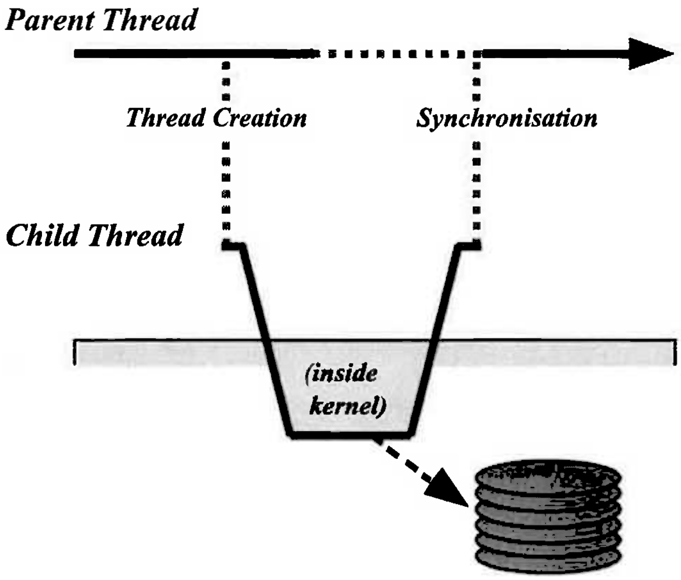

Figure 10 – Using a Thread for Asynchronous I/O.

has been scheduled[27] in the case of a write operation. Importantly, this information is not actually used in the child. Instead it is passed back to the parent and used there.

The use of a child thread for asynchronous I/O is shown in Figure 10. This approach is riddled with problems, as described below:

- ***Loss of Thread Isomorphism*** – The child thread in the implementation domain does not correspond to a computational activity in the design space – the child thread is just an artefact of the implementation. This is itself an immediate reduction of traceability. Worse, revision of the system may lead to extra code being added to the child thread. This causes the child thread to contain real computational activity. However, this computational activity was never part of the design, and the correspondence from design space activities to implementation space threads has been well and truly destroyed, amplifying the adverse effect on system traceability and thereby increasing the difficulty of construction and maintenance of a correct implementation.
- ***Inappropriate Abstractions*** – The previous problem derives from the fact that the abstraction used in the implementation, the thread, is inappropriate for the task. What is genuinely required is to provide the parent with an abstraction, which we shall call *AsynchIO*, with two operations:
  - *schedule(...)* – starts the operation, and
  - *sync()* – synchronises with the completion of the operation.

  At the level of this abstraction, the implementation avoids the need for an artefact thread. Within the abstraction, a thread may be used as the basis for implementation, however in this case the abstraction is still subject to the problems listed in the remainder of the list.
- ***Inappropriate Thread Granularity*** – Even if the child thread is hidden inside an abstraction, the fact remains that it does very little computation, the thread is therefore has very fine granularity. However, because the child thread executes a system call, it must have kernel support, so it cannot be implemented using the finest grain threads mechanism, pure user-level

27 In some operating systems it may be possible to cause write operations to block until the data has actually been written to disk. For example, Unix allows the O_SYNC flag to be set on file descriptors.

threads. In fact, even user-level threads are too coarse in granularity. User-level threads provide each thread with a private stack and a complete register file. Neither of these is needed in the implementation of the abstraction. However, both must be maintained by the threading mechanism since the thread's context must be swapped out after it issues the I/O operation and back in after the I/O operation completes.

- ***Poor Performance*** – Using a thread as the basis for implementation of the *AsynchIO* abstraction is inefficient. Some of this inefficiency derives from inappropriate thread granularity. A more significant inefficiency arises when a single parent thread creates several *AsyncIO* instances to run in parallel. Each of these will execute at least one system call, with the associated overhead of trapping to the kernel. In contrast an implementation making use of the asynchronous I/O facilities defined under POSIX can submit several I/O operations in a *single* system call using the *lio_listio()* function[28].
- ***Unexpected Semantics*** – Even if the previous problems are avoided, the I/O operation performed by the thread-based implementation of the *AsyncIO* abstraction is likely to exhibit different semantics to a synchronous I/O operation. I/O operations on files, such as *read()* and *write(),* implicitly operate on the location in the file pointed to by a process-wide *read/write* pointer. Furthermore, the operations change this pointer. If a (parent) thread creates several child threads to perform concurrent I/O, these child threads can execute in any order, with the result that read and writes might operate on the "wrong" part of the file. One way to solve this problem is for the threads implementing the I/O operations to be interlocked so that the operations execute in the order that they were submitted. Clearly, this requires the addition of a synchronised queue accessible to all instances of the *AsynchIO* abstraction. It also eliminates any concurrency between I/O operations, and in that case, there is no need to bother with one child thread to each of the *AsynchIO* instances applying to a given file – one will suffice for all. The result is that there is a single thread responsible for all *AsynchIOs* for a given file. This thread performs significant computation and hence avoids the design and other problems with the thread-per-operation approach. In effect, the problem has been defined out of existence. The alternative is for the operating system to provide versions of the *read()* and *write()* system calls which include the location within the file as an explicit parameter, but this is a more complex interface than that presented by the standard *read()* and *write().* The POSIX asynchronous I/O facilities include the location within the file for every read and write operation.

Clearly, the supposedly straightforward approach to asynchronous I/O based on child threads is more trouble than it is worth. A specialised asynchronous I/O facility is just as easy to use as the thread based mechanism.

Another common abuse of threads is to perform asynchronous remote procedure calls, in much the same way as the approach just described was used for asynchronous I/O. Again, this has problems:

- ***Loss of Thread Isomorphism*** – the design requires the submission of an RPC, and later synchronisation with the reply, but the implementation contains a thread. Again, threads are an inappropriate abstraction.
- ***Inappropriate Thread Granularity and Poor Performance*** – for an RPC over the network, the child thread does little other than perform system calls. It might be made to do the marshalling of the parameters to the RPC, however this may become difficult since the parent thread may

[28] This is assuming that *lio_listio()* is implemented as a single system call. POSIX compliant systems could implement it as a collection of single asynchronous I/O system calls, or even as a collection of synchronous I/O system calls in a separate thread! The minimum kernel support for POSIX asynchronous I/O are versions of *read()* and *write()* that allow explicit specification of the position within the file to perform the operation.

concurrently mutate those parameters. There is however, no such problem in the unmarshalling of the result.

Almost certainly, one is better off with an abstraction similar to *AsynchIO* for asynchronous RPCs rather than were one simply to use threads. For RPCs over a network, this abstraction can be implemented more efficiently using a smaller number of threads and asynchronous (network) I/O system calls. Conversely, on some operating systems, for RPCs between processes on the *same* machine, creating a thread to execute the call may provide better performance. Some modern operating systems, such as Spring [Hamilton and Kougiouris 1993] and Grasshopper [Rosenberg *et.al.* 1996], provide the ability for a thread to, in effect, jump between address spaces (processes). In this case, it makes sense for a (child) thread to be created on the client. This child thread then jumps to the server, executes the remote operation, and then returns to the client with the result of the call, performing unmarshalling and synchronisation after the return. That the local and remote case should have such disparate implementations strengthens the case for an abstraction.

In summary, threads are designed as a mechanism to express asynchronous computation, They are not for expressing asynchronous I/O nor for asynchronous remote communication. It is advisable to restrict their application to that for which they were intended.

2.4 Heterogeneity Considerations

Heterogeneous distributed systems incorporate nodes that are in some way disparate. Several factors result in heterogeneity in typical distributed systems:

- ***Use of Existing Hardware*** – it is rare for a distributed system to be deployed entirely on new hardware. Instead, all or part of the system is deployed on existing hardware. This hardware is likely to be of several different kinds. This is particularly likely to be the case if an existing distributed system is being expanded to incorporate an additional (possibly remote) group of users.
- ***Use of Legacy Components*** – a distributed system may require access to data or services held in an existing (legacy) system. It is usually the case that the most cost-effective strategy for providing this access is to incorporate the legacy system into the new system as a component, instead of developing a new component with equivalent functionality. The use of legacy components is likely to lead to linguistic heterogeneity. For example, the legacy component may have been developed several decades previously, in which case it is unlikely to have been implemented in the programming language chosen for the new part of the system.
- ***Use of Specialised Facilities*** – specialised components of the system may require specialised implementation strategies. For example, an image-processing component of a document automation system may be implemented on specialised hardware, such as an array processor, whereas the remainder of the system may be implemented on hardware that is more conventional.

These considerations have great deal of force in today's distributed systems. In the distributed systems of the future, operating over the Internet or corporate intranets, where systems developed by completely different entities may need to be swiftly integrated, heterogeneity will be even more important.

Three main kinds of heterogeneity can be identified. All or any combination of these may simultaneously be present.

- ***Architectural Heterogeneity*** – where nodes of the system execute on different processor architectures.
- ***Operating System Heterogeneity*** – where nodes of the system are running different operating systems.

Byte Order		**Conversions Required**	
Sender	*Receiver*	*Network Byte Order Scheme*	*Flag Scheme*
Network	Network	0	0
Network	Other	1	1
Other	Network	1	1
Other	Other	2	0

Table 4 – Byte-order Conversions.

- ***Linguistic Heterogeneity*** – where nodes of the system are implemented in different programming languages.

Supporting these three kinds of heterogeneity has a considerable impact on the communication mechanism at the language level. In particular, heterogeneity significantly constrains the communication mechanism design. The impact of the different kinds of heterogeneity is discussed below.

2.4.1.1 Architectural Heterogeneity

Architectural heterogeneity arises from differences in processor architectures. It has two main facets:

- Different hardware architectures may represent a given kind of data in different ways.
- Different hardware architectures contain different instructions sets.

In modern practice, the first facet is primarily a concern related to the internal implementation of a communication mechanism. The second facet, however, actually constrains the freedom available in the communication mechanism design.

Variations in data representation can occur in several forms, the most common of which is a variation in the order of the bytes in multi-byte data such as integers. The two most common alternatives are called *little endian* and *big endian*. This is no good reason why there should not be a single standard byte-order, but because there are now so many processor architectures in each category, a problem of architectural heterogeneity occurs in practice and will continue in the foreseeable future.

This is commonly addressed in one of two ways. One way is to define one of the alternatives as "network byte-order" and require nodes executing on hardware which does use network byte-order natively to convert all data sent and received over the network. The other way is for a node sending data to transmit it in native format and include a flag specifying what that native format is. The receiver of the data then checks the flag and converts the data only if it is necessary to do so. Inspection of Table 4 indicates that the flag-based alternative is to be preferred, due to the reduced number of conversions required. Of more importance is the fact that the task of coping with this kind of heterogeneity can (and should) be done internal to the implementation of the communication mechanism; hence there is no impact at the programming language level.

Some other examples of data representation heterogeneity include the use of different sizes of bytes and the use of different methods of encoding for various data types. Once again, these can be managed entirely within the communication mechanism and, thus, are invisible at the language level.

In contrast to variations in data representation, variations in instruction set and instruction representation may have an impact at the language level. In general, it is impossible to send compiled object code for a given architecture to a node with a different architecture to be executed there. This makes it difficult to support the transmission of functions, procedures and other code-containing entities in network messages at the language level. By far the most common strategy is simply to prohibit the sending of such entities through the communication mechanism – in this case, the programming language level is constrained by heterogeneity.

In contrast, the strategy adopted in Java is to send *partially compiled code* over the network. When this code arrives at the remote node, it may be executed via interpretation. Alternatively, a *just-in-time* (*jit*) compiler on the remote node may be used to complete the compilation to (remote node) native code that is then executed directly by the hardware. These strategies are both effective in abstracting over variation in instruction set and thus removing any impact on the language level. The interpretation-based approach is proven but it executes significantly slower than code produced by an optimising compiler. The just-in-time approach offers the possibility of execution speeds very close to those produced by a normal compiler, however the cost of running the *jit* compiler during system execution must also be taken into account.

Another strategy is to emulate the execution of the instructions received over the network in software. This strategy has been successfully used in the execution of M68000 Macintosh binaries on PowerPC Macintoshes, on Sun and Hewlett-Packard workstations running Unix, and in the execution of Microsoft Windows' binaries on a variety of different hardware. Execution of emulated code is an order of magnitude slower than the execution of native code. Some strategies are able to provide an improvement on this, for example it is possible to compile parts of the foreign code into native code. Nevertheless, the foreign code will still execute several times slower than native code. Furthermore, this approach does not really scale to a large number of different architectures – for n architectures to interoperate using this approach, a total of n^2 emulators will need to be developed.

2.4.1.2 Operating System Heterogeneity

Operating systems heterogeneity presents similar challenges to those presented by architectural heterogeneity. Again, operating systems exhibit two facets of heterogeneity:

- Different methods of encoding might be used for a given data type.
- Different environments might be provided for program execution.

An example of the first case is the use of ASCII versus EDCDIC for the encoding of character data. As was the case for architectural variations in data representation, this difference can be entirely abstracted away by the communication mechanism.

Differences in execution environment primarily consist of variations in the system calls and standard library functions provided by the operating system. This can be dealt with successfully using a strategy consisting of the components listed below.

- The ubiquitous use of dynamic linking for interfaces provided by the operating system.
- The provision of dynamically linked libraries to emulate facilities provided by foreign operating systems.
- Zero or more privileged processes, as required, to enable the emulation libraries to provide the exact semantics of the foreign operating system.

It is quite feasible to support execution of binaries from a foreign operating system in this way. For example, dynamically linked SunOS UNIX binaries can be executed under the Spring operating system [Khalidi and Nelson 1993], via the use of a shared library (*libue.so*) and a privileged process providing certain Unix specific functionality such as signals.

In summary, the task of producing a production quality implementation of a communication mechanism faces a number of challenges in supporting operating system heterogeneity. However, these challenges can be answered successfully given sufficient resources. Furthermore, they can be answered in a way that has no impact on a communication mechanism *at the language level*. Consequently, this thesis need not devote further attention to this issue.

2.4.1.3 Linguistic Heterogeneity

Linguistic heterogeneity occurs in a distributed system that incorporates computational elements implemented in two or more different programming languages.

It is obvious that linguistic heterogeneity can arise in relation to legacy components of new systems. A legacy component will have been implemented in a language that seemed appropriate to the developers at the time. Large systems contain multiple legacy components, typically implemented in several different programming languages, so there is often linguistic heterogeneity between legacy components, without even considering new components. Even if all the legacy components are implemented in the same language, this is likely to be unsuitable for development of new components, particularly if the legacy components are old, as many are. Consequently, there is linguistic heterogeneity between the new components and the legacy components, at the very least.

This being said, the role of legacy components in the system is limited. Legacy components provide services to new components, they do not typically obtain services from new components or from other legacy components. Consequently, the mechanism for communication between components does not need to support communication originated from legacy components. A system should not be denied linguistically heterogeneous status simply because the communication mechanism is inaccessible from the programming languages used to implement one or more legacy components.

It is also likely that there will be linguistic heterogeneity between new components, in addition to that between legacy components and between new and legacy components. There are two main sources of such heterogeneity:

- The use in the system of off-the-shelf components, bought rather than built to reduce development cost. These might be implemented is programming languages that are different to those chosen for locally developed components.
- Organisational mergers may lead to the need to merge multiple existing systems whilst continuing to maintain and improve the functionality of each existing system as a sub-system of the merged system. These sub-systems can not be considered legacy components, as they are subject to continuing development, and will often have been implemented in differing languages. Such existing systems are usually sufficiently large as to make porting on sub-system to the language used for another impractical.

Consequently, linguistic heterogeneity is often unavoidable, including linguistic heterogeneity between new components.

This thesis assumes that non-legacy components are implemented using strongly typed programming languages. Furthermore, it is assumed that non-legacy components make use of legacy components but not the reverse. Legacy components do not make use of non-legacy components, though they may use other legacy components[29]. Consequently, the type-security of the non-legacy part of the system can be assured by encapsulating (type-unsafe) legacy components inside strongly typed *wrappers* that perform the necessary validations. One of the major activities of the Object Management Group (OMG) is the definition of standard wrapper interfaces encapsulating a variety of legacy system components. Finally, it is assumed that code does not migrate across linguistic boundaries. From a technical standpoint, converting between different instructions sets with relatively simple execution semantics is a difficult task, converting between different high level languages with far more complex semantics is sufficiently difficult as to be considered infeasible in practice.

These assumptions imply that the issue of linguistic heterogeneity in general degenerates into an issue of type-system heterogeneity. Thus, it is the type-system of a language, rather than the language

[29] This communication between legacy components is invisible to non-legacy components.

itself, which is the unit of linguistic heterogeneity. So a linguistically heterogeneous distributed system is simply a system in which different parts are governed by different type-systems.

Heterogeneity and the Type System

The use of message passing for communication is a virtual necessity if type-system heterogeneity is to be supported. Sharing based communication permits situations where a given region of the shared address space is simultaneously subject to the jurisdiction of several type-systems and language implementations. This is overly constraining on both type-systems and language implementations. In contrast, with message-passing the points of tension between type-systems are easily defined as message send and receive operations, any necessary validation can take place at these points. Furthermore, since the send and receive operations intrinsically involve the copying of data at the conceptual level, any transformation of the data required by the interacting type-systems can be performed as part of those operations. As discussed previously the previous chapter, support for linguistic heterogeneity is one of the significant advantages of message passing over distributed shared memory.

Given that strongly typed languages communicate via message passing, this message passing also needs to be strongly typed. There is no point going to the trouble of providing type security within a given node only to have it compromised by the arrival of a message which contravenes the node's type system.

Given that nodes in the system communicate via strongly typed message passing, a language is needed for the expression of that communication. There needs to be a *consensus communication type-system* with jurisdiction over messages between any given pair of language type-systems. Clearly, for a given consensus communication type system to be effective, it needs to be acceptable to each of the languages type-systems which send and receive messages subject to its jurisdiction.

In summary, three requirements for effective support of type-system heterogeneity in distributed systems have been identified.

- Message-passing communication.
- Strong typing of messages.
- The existence of a consensus communication type-system with jurisdiction over the messages sent between any pair of language type systems, which is acceptable to both language type-systems.

The most obvious example of a consensus type system for communication is the interface definition language (IDL) defined by the OMG in the CORBA 2.1 specification [OMG 1997].

Taken together, the three requirements listed above lead to a class of software systems termed *strongly typed middleware*. The role of the middleware sub-system in heterogeneous distributed systems is to act as a *request broker* providing communication between elements of the system. Programming languages in the system submit requests and retrieve replies via a language specific binding to the middleware IDL. The binding mechanism would include a stub compiler that translations the middleware IDL into language specific stubs, thus preserving type safety (assuming the implementation language is type safe). At the server end, the middleware request broker accesses a language specific adaptor to deliver requests to operations defined in a particular programming language.

CORBA is the most obvious example of a strongly typed middleware system specification. CORBA binding and adaptors have been defined for a number of well known programming languages, including C, C++ , Smalltalk, Cobol, Fortran, Java and Eiffel.

2.5 The Case for Language Level Middleware

The issues addressed in previous sections all have implications at the programming language level.

- ***Message-passing Mechanisms*** – Easy-to-use message-passing mechanisms, such as RPC, require a language for specifying interfaces and require binding between that interface definition language and programming languages used in the implementation of the system.
- ***Software Engineering*** – The coding phase of software engineering is clearly dependent on the programming language. It may be the support that a programming language provides for aspects of modern software engineering practice that makes it the preferred tool for a given task. Alternatively, it may be something to do with the application being implemented, for example, mathematical programs are implemented in Fortran partly because of the availability of powerful mathematical libraries.
- ***Heterogeneity*** – The approach to heterogeneity discussed above is dependent on type-systematic information, this is a language level issue.

In combination, these factors lead to the conclusion that supporting message-passing communication in distributed systems with the simultaneous goal of software engineering and heterogeneous systems support must be addressed by middleware providing strongly-typed interfaces at the programming language level.

Having established that language level support is required, the next task is to determine the language paradigm that should be adopted in the middleware (most particularly, in its IDL). This thesis adopts object technology as the language paradigm, for three main reasons:

- The majority of modern software engineering techniques are (at least partially) object based.
- Object based techniques provide a structure for the incorporation of programmer controlled concurrency into a system. Previous techniques for software engineering do not provide this general support for structuring concurrency.
- Interactions between objects are commonly described as message passing at the conceptual level, this maps directly to message-passing at the physical level, as is required in the approach to distributed systems taken in this chapter (and in this thesis).

Object technology and distributed object systems are discussed in detail in the next two chapters of this thesis.

Chapter Three

Object Technology for Sequential Systems

This chapter presents a discussion of the current state of the art in object technology in sequential systems, as a precursor to the extension of the object technology model to concurrent and distributed systems in the next chapter. The focus of this thesis is on the middleware between objects, rather than on the implementation of the objects, so this model of object technology concentrates on the interfaces between objects, rather than their implementation. The object technology model is used as the basis for the remainder of this thesis.

Object technology, so long in gestation in the '70s and '80s, is finally coming to fruition in the '90s. This thesis is concerned with object technology at the infrastructure level rather than the application development level. This infrastructure level includes programming language and run-time support issues. Object technology infrastructure can be divided into four kinds of facilities:

- The core object model; this is described in Section 3.2.
- Facilities for abstraction and information hiding, as described in Section 3.3. These are further developed in the notion of contract, as described in Section 3.4.
- Inheritance facilities; these are described in Section 0.
- Object management facilities, as described in Section 3.6.

There is a particular focus on the notion of contract governing the validity of methods calls, including the use of exception to handle invalid calls. The importance of this in the context of the thesis arises because inter-module communication in sequential object systems occurs via method calls. In distributed object systems, communication is also effected via method calls. The notion of contract developed in this chapter is applied to concurrent and distributed object systems in the remainder of the thesis.

3.1 Note on the Use of Eiffel Syntax

In this chapter and the next in particular, and more generally in the entire thesis, extensive use is made syntax following that of the Eiffel[30] [Meyer 1988, 1992, 1997] programming language in the presentation of programming examples. This is because the Eiffel approach to object-orientation is the closest among well-known languages to the approach advocated in this thesis. The examples are not Eiffel, but are instead intended to be illustrative of various object-oriented concepts. In general, the differences from Eiffel syntax can be classified as follows:

- Differences where the equivalent Eiffel syntax is sufficiently idiosyncratic as to require detailed explanation. In these cases, syntax that is more conventional is adopted.

[30] It is amusing to note that the name Eiffel has been excised from the second edition of *Object Oriented Software Construction* [Meyer 1997]. In that edition, it is replaced by the phrase "the notation used in this book". This thesis could take this practice to an extreme: "the notation used in [Meyer 1997]", but the term Eiffel is more convenient.

- Differences where semantic concepts not present in Eiffel are introduced.
- The use of "-" in identifiers, where Eiffel (and most other languages) would have "_". This is due to typographical issues in the production of the thesis.

Such syntactic differences are noted at the first point of the text that they appear.

3.2 Core Object Model

There are many definitions of the term "object" in circulation. In the view presented in this thesis, objects consist of *state*, *behaviour* and *identity*.

The state of an object consists in zero or more instance variables. Instance variables resemble fields in a record (Cartesian product) data type. Each instance variable may be referenced by name and the lifetime of the set of instance variables comprising the state of an object is the lifetime of the object as a whole. For example, the state of a *BANK-ACCOUNT* object might include an instance variable *balance* which is a fixed-point decimal number providing two decimal digits of precision.

The behaviour of an object is defined in terms of operations or methods. Operations are typically defined in a similar way to functions. In contrast to pure functions, the result(s) of operations may depend on the values of instance variables of the object being operated on. Furthermore, operations may mutate the state of the object that is operated on. In fact the only result of an operation might be the mutation of the state. For example, a *withdraw* operation on a *BANK-ACCOUNT* object might simply decrease the *balance* instance variable by the required amount. The most common syntax for demoting the invocation of an operation is the "dotted procedure call" or "remote call", for example, if a *BANK-ACCOUNT* object is called *my-account* and it is desired to withdraw $50.65, one would write:

```
my-account.withdraw(50.65)
```

This syntactic convention is unimportant. In Smalltalk, one would write:

```
my-account withdraw : 50.65
```

and in Ada 95, one would use conventional procedure call syntax, as in:

```
withdraw(my-account,50.65)
```

Within the implementation of an operation for a given object, it is possible to invoke other operations on that object without explicitly specifying the object as the target. For example, the call:

```
withdraw(50.65)
```

in the implementation of one of the *BANK-ACCOUNT* object's operations invokes the *withdraw* operation on the object for which the operation containing the call was invoked.

The third element in the core object model is the notion of identity. Identity provides the ability to distinguish two (distinct) objects even if they happen to have the same operations *and* the same state values. For example, objects representing two people, Larissa and Henry, might both include a *BANK-ACCOUNT* object as an instance variable. These *BANK-ACCOUNT* objects have the same operations and might have the same state values at some point in time. However, even when the state is equal, the two are not the same object, and in particular, a *withdraw* operation executed on the *BANK-ACCOUNT* belonging to Henry should not effect the *BANK-ACCOUNT* belonging to Larissa.

Object identity must also support sharing where required. Suppose that Henry and Larissa were to marry, and to establish a single, shared bank account. In this case, the state of the objects representing Larissa and Henry should contain references to a single shared *BANK-ACCOUNT*, so that operations on either instance variable would be reflected in both. Object identity also supports cyclic relationships: two objects can have (possibly indirect) references to each other's identities.

It should be noted that in this core conception of an object there is no protection of the state of an object from direct access from outside. For example, a dishonest user of a *BANK-ACCOUNT* object could add a million dollars to the balance by direct assignment to the *balance* instance variable:

```
my-account.balance := my-account.balance + 1000000.00
```

This is clearly undesirable, the prevention of such abuse is a motivation for facilities for object-based abstraction, which is now discussed.

3.3 Abstraction and Information Hiding Facilities

A fundamental principle of software engineering is the hiding of complexity, also termed *information hiding*. Parnas [Parnas 1972], discusses an approach to modular decomposition that is driven by information hiding:

> *"The second[31] decomposition was made using "information hiding" as a criterion. The modules no longer correspond to steps in the processing. ... Every module in the second decomposition is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings."*

In other words, the complexity of the implementation of a module should be hidden behind the relative simplicity of a well-defined modular interface specification. Clients of the module interact with the module only via the interface, thus using abstraction to avoid any need to consider the implementation details. This use of abstraction has several additional advantages, including the following:

- De-coupling of the implementation from the interface. This allows the implementation to be improved without affecting the module's clients.
- Protection of the state data of the implementation from unauthorised tampering of the kind noted previously.
- Freeing the implementor of the module to implement the specification of the module given in the interface, rather than being beholden to an ever-changing and *ad hoc* notion of client requirements.

In object technology; objects are the modules. There are four kinds of facilities for providing object-based abstractions in general use:

- Data abstraction.
- Classes and types.
- Sub-typing and substitutability.
- Genericity and parametric polymorphism.

These are discussed in Section 3.3.1 to 3.3.4 respectively.

3.3.1 Data Abstraction

Data abstraction is the protection of the state of an object so that it may only be accessed via the object's defined operations. To return the *BANK-ACCOUNT* example, the abuse perpetrated by the dishonest user would be prevented by exporting an interface including operations such as *withdraw*, *deposit* and *transfer* and rendering the state information inaccessible except via this interface.

3.3.1.1 Commands and Queries

Two kinds of operations are available on objects:

[31] Parnas also presents a functional decomposition of the same system to provide a basis for comparison.

- *Commands* – which do something to the object, usually changing its state variables in some way.
- *Queries* – which return some information about the current state of the object.

At the programming language level, commands are represented by procedure methods and queries by function methods.

Functions in the functional programming sense do not produce side effects. All expressions in a purely functional programming language have the property of referential transparency, defined by Meyer for the object-oriented context as follows:

> *"An expression e is referentially transparent if it is possible to exchange any subexpression with its value without changing the value of e"*[32]

Referential transparency assists reasoning about the correctness of software. Programmers typically have had a decade or more exposure to mathematical reasoning in school and higher education. Expressions in mathematics are referentially transparent, so software expressions that are similarly referentially transparent enable programmers to employ well understood mathematical reasoning approaches to the task of reasoning about their software.

Consequently, it is desirable to find some way to include referential transparency in object-oriented programming. No constraints need be specified on commands since they may not form parts of expressions. Queries however, may be part of expression, and therefore need to be constrained so as not to produce side effects that violate referential transparency.

One way to enforce referential integrity is to constrain queries so that they do not produce side effects. This constraint may be enforced by prohibiting the following operations in the code of queries:

- Calls to commands on the object queried, on an object passed as a parameter, or any object in the transitive closure of the state variables of those objects. Syntactically this means that the only valid command calls are those applied to a target beginning with a local variable. For example:

  ```
  lo.aquery(params…).bquery(params…).ccommand(params…)
  ```

  where *lo* is a local variable in the query containing this call.
- Assignments to any of the objects state variables, to any state variable of an object passed as a parameter, or to any state variable of objects in the transitive closure of those state variables. Syntactically this means that the only valid assignments are those applied to an L-value expression beginning with a local variable[33]. For example:

  ```
  lo.aquery(params…).bquery(params…).cvar := 79
  ```

  where, again, *lo* is a local variable in the query containing this call.
- Assignments that would result in the object, any objects passed as parameters, or any object in the transitive closure of those objects becoming reachable[34] in the transitive closure of the query's local variables. Permitting such assignments could allow assignments permitted by the previous rule to indirectly mutate the object's state or that of objects in the transitive closure of

[32] This definition is given on page 750 of [Meyer 1997].

[33] In the Eiffel language, the only assignments permitted under any circumstances are to the state of the current object and to the local variables of a method. In this case, it is sufficient (and simpler) to prohibit any assignments to the object state in queries.

[34] Actually, the problem is not that such object become reachable, but rather that they become reachable in a form that allows them to be mutated. In language such as C++, in which references can be marked as not supporting mutating operations (**const** in C++), it is possible to allow the assignment providing the L-value is a marked as not supporting mutating operations.

that state. Syntactically this means that the R-value expression for an assignment within a query should not begin with a state variable, or with the identifier used to denote the current object.

Notice that queries can call other queries, this is acceptable because those queues are themselves subject to the above constraints. Note also that there is no restriction on the creation and return of new objects by the query.

3.3.1.2 Abstract and Concrete State

The preceding rule ensures referential transparency. However, it is in fact more constraining than is necessary or desirable in practice. Consider a class that provides random access to a file resident on disk. This class would provide a query that returned the contents of a particular location in the file. In principle, this query could be implemented by reading the appropriate disk block each time the query is called (given that the operating system provides a read operation can be considered a query). In practice, however, it would be advisable to cache data that has been accessed recently, to take advantage of locality in file access and reduce the number of disk operations. The code of the query might therefore be something like:

```
read(location:INTEGER): BYTE is
  local
     block : INTEGER := locToBlock(location)
  do
     if not cache.contains(block) then
        cache.install(os.readblock(block))-- command to update cache.
     end
     Result := cache.lookup(location)
end
```

This includes a command to update the cache if the required location is not found. This command is illegal under the rule given previously, but it does not violate referentially transparency of calls to *read*.

This example demonstrates a need to distinguish between the concepts of abstract and concrete state. In the example, the concrete state of the object includes the cache, but the abstract state is the current context of the entire file. The query given above does not alter the abstract state, but does alter the concrete state. Given information hiding is in force, an object's concrete state is a purely internal representation; only the abstract state is accessible from outside the object. Therefore, only mutation of abstract state violates referential transparency for the external observer. Consequently, the restriction given previously is replaced by a methodological guideline prohibiting queries from mutation abstract state but permitting the mutation of concrete state.

3.3.2 Classes and Types

Given that significant time is invested in the development of objects such as *BANK-ACCOUNT* it often useful to be able to create multiple instances of such objects. These instances would provide the same operations and state variables (though with independently varying values). The notion of a *class* provides this facility. A class acts as a template from which objects of a given kind may be created. The class of an object defines the operations associated with that object and the names of its state variables. The class may also define the initialisation of those state variables for newly created objects. It is immediately apparent that *BANK-ACCOUNT* ought to be a class. This is because it is almost certain that there will be more than one bank account in a system and there is no point defining the operations and state of each of the objects separately. An initial versions of the *BANK-ACCOUNT* class is shown in Figure 11.

```
class BANK-ACCOUNT
   feature {NONE}
      balance
   feature
      create is …
      deposit(amount) is …
      withdraw(amount) is …
      transfer(to-accnt,amount)is …
      get-balance is …
      get-min-balance is …
end
```

Figure 11 – BANK-ACCOUNT Class.

```
class BANK-ACCOUNT
   feature {NONE}              -- private members
      balance : decimal
   feature                     -- public members
      deposit(amount : decimal) is …
      withdraw(amount : decimal) is …
      transfer(to-accnt : BANK-ACCOUNT,amount : decimal) is …
      get-balance : decimal is …
      get-min-balance : decimal is …
end
```

Figure 12 – Strongly Typed BANK-ACCOUNT Class.

As discussed in a previous chapter (Section 2.3.1.2), strong typing is an almost essential tool for the software engineering of production systems. The type of a value or variable is simply a specification of the operations applicable to that entity. The two terms type and class are defined as follows:

Type – a specification of the operations applicable to an entity.

Class – an implementation of a (user-defined) type.

Note that these definitions admit the possibility that there are several classes implementing a given type. This possibility is apparent in the Theta language [Liskov et.al. 1995], which linguistically separates the two notions. Most strongly typed object-oriented languages such as Eiffel [Meyer 1988, 1997] and C++ [Stroustrup 1991, ISO 1998], do not have complete linguistic separation; nevertheless, the distinction is still present in the underlying semantics. This thesis adopts this second approach, a strongly typed version of the *BANK-ACCOUNT* class is shown in Figure 12.

The type of the object may be simply the aggregation of the types of the operations, in this case type equivalence is structural. More commonly, the name given in a class definition is also significant, in which case type equivalence is name based. In either case, the type of a class describes only the interface used to interact with objects of that class. It should not include the types of the class's instance variables, these are not part of the interface and hence may vary as the implementation is refined. The main difference between classes and types is that (object) types are purely abstract specifications of interfaces, whereas classes may include object state and operation behaviour in addition to operation interfaces. In addition, entities other than objects may be typed (but may not be members of classes). Some examples include basic types such as integers and types for operation interfaces (analogous to procedure and function types in their respective programming paradigms).

3.3.2.1 Constructors and Other Special Methods

The ordinary methods defined by a class implement the operations permitted on instances of that class. A constructor is a special method responsible for transforming a region of memory allocated to a new class instance into a valid class instance, observing any invariants on the state defined by the programmer. Constructors may take parameters, the actual values passed to these parameters may be used in the initialisation of the state. A class may define more than one constructor, in which case each constructor must have a parameter list differing in number or types of parameters. A class may also omit definition of a constructor, in which case the system will generate a default constructor that initialises the state to language defined default values.

In addition to constructors, some languages allow the definition of destructors. Destructors are methods that are called prior to the deallocation of class instances. Destructors are called implicitly, (by the language system) and consequently they do not take parameters, and there may be at most one destructor defined for a given class.

Finally, some languages allow the programmer to supply the implementation of the assignment operation for objects of a class.

3.3.2.2 Class-wide Methods and Data

Ordinarily, method calls apply to an instance of a class, and similarly, state is associated with a particular instance of a class (each instance has its own copy of the state). It is also possible to support both methods and state that are associated with the class as a whole. These entities, which are associated with a class as a whole, rather than with a particular instance, are referred to as class-wide (methods or data). There is a single copy of any class-wide variable, this is shared between all the class instances. Class-wide methods may access only class-wide data (ordinary methods may access both class wide and instance-specific data).

Java is one language supporting class-wide data and methods. In addition, Java supports the definition of class-wide constructors. These are parameterless sub-programs responsible for initialisation of the class wide data. Class-wide constructors are called implicitly by the language system (in Java this occurs when the class is loaded).

3.3.3 Sub-typing and Substitutability

Another facility for supporting abstraction is the notion of polymorphic assignment and operation invocation, also known as dynamic binding or sub-typing. Without this polymorphism, a variable declared to hold a particular type are only permitted to contain objects that are instances of the class defining that type. Polymorphic assignment relaxes this rule. It permits assignments in which the R-value type, *S*, *conforms to* the L-value type, *T*. In such a situation, *T* is called the declared type of the variable being assigned to and (after the assignment) *S* is the dynamic type of the value of the variable. Multiple polymorphic assignments may cause the dynamic type of a given variable to vary during its lifetime. Polymorphic operation invocation ensures that, where an operation *O* is invoked on a value with declared type *T* and current dynamic type *S*, the version of *O* defined by the class of *S* is executed.

Conformance (also known as substitutability) means that an object of a class, *S*, may be attached to a reference of type *T*, provided that *S* conforms to (is substitutable for) *T*. This attachment may occur via assignment or parameter transmission. Informally, anything that can validly be done to an object can also validly be done to an object of a type conforming to that object's type. Given data abstraction, the only valid actions on an object are invocations of that object's operations. The specifications of these operations comprise the object's type. Hence, one can also say that a given class conforms (or does not conform) to a given (abstract) type.

Classes automatically conform to their own type. In a system with structural type equivalence on the aggregation of operation types, any two classes with interfaces declaring the same operations conform to each other. In the more common named based equivalence regime, a class must be explicitly declared to conform to the type of another class in addition to providing a matching interface. For example, a *PRIVELEGED-BANK-ACCOUNT* could be defined and declared to conform to the *BANK-ACCOUNT* class. The implementation of the *withdraw* operation for *PRIVELEGED-*

```
class PRIVELEGED-BANK-ACCOUNT conforms35 BANK-ACCOUNT
   feature {NONE}
      balance : decimal
      overdraft : decimal
      ...
   feature
      deposit(amount : decimal) is ...
      withdraw(amount : decimal) is ...
      transfer( to-account : BANK-ACCOUNT,
                amount : decimal) is ...
      get-balance : decimal is ...
      get-min-balance : decimal is ...
      get-overdraft : decimal is ...
end
```

Figure 13 – Privileged BANK-ACCOUNT Class.

BANK-ACCOUNT would be redefined to allow the use of an overdraft facility. For the purposes of deducting monthly bank changes, a list of *BANK-ACCOUNT* variables is maintained which contains objects with both *BANK-ACCOUNT* and *PRIVELEGED-BANK-ACCOUNT* dynamic types. However, all constituent objects are operated on in exactly the same way:

```
current-account.withdraw(4.50)   -- Deduct monthly fee
```

and the polymorphic operation invocation mechanism executes the version of the *withdraw* operation appropriate the invoked object's dynamic type. Thus, this polymorphism mechanism permits an additional degree of abstraction: abstraction over the class implementing a variable of a given type. A common use of this mechanism involves the definition of an abstract class declaring a given interface, which does not provide any implementation of the operations in the interface, and several concrete classes (conforming to the type of the abstract class) providing alternative implementations.

Conformance can be made more general; instead of requiring that a conformant class implement exactly the interface of the type conformed to, the class may be permitted to include additional operations in its interface, providing the operations in common with the class to be conformed to have the same interface. For example, the *PRIVELEGED-BANK-ACCOUNT* class shown in Figure 13 adds a *get-overdraft* operation to return the amount that the account may be overdrawn.

Since an exact match is no longer required, a class may also be declared to conform to several classes. Furthermore, the conformance relation is transitive, but the graph of conformance relationships thus constructed must be acyclic.

The rules for conformance may be relaxed still further, by allowing a conformant class to change the return type of an operation to be a type conforming to the return type declared in the class conformed to (covariant result definition). However, redefinition of the argument types of operations in the same way (covariant argument redefinition[36]) is, in general, unsound[37], although practical solutions have been developed.

The facilities for abstraction that have been presented thus far are a fairly straight-forward development of type checking from the current set of typed imperative languages, coupled with facilities for encapsulation and for polymorphism. To enable the construction of more effective abstractions, the specification given in the interface of an object must cover more than just type

[35] The ***conforms*** reserved word is not Eiffel syntax, it expresses a semantic relationship between types that is independent of an inheritance relationship between classes implementing those types, something not available in Eiffel.

[36] The alternative, contravariant argument redefinition allows the types of arguments to be redefined to types that they conform to. This is intrinsically sound, but mostly of limited utility.

[37] See [Cardelli and Wegner 1985] for a discussion of argument redefinition.

checking of operations. Section 3.4 considers specifications as *contracts* between the implementor of an abstraction and the abstraction's users. Initial consideration is as an alternative view of the abstraction and typing mechanisms described above. Subsequently this will be developed as a means of extending specifications into the semantic arena.

3.3.4 Genericity and Parametric Polymorphism

The facilities for sub-typing and substitutability introduced above allow the creation of software components that abstract over the precise type of values operated on. Consider a type *SHAPE* with sub-types *SQUARE* and *CIRCLE*. Given substitutability of sub-types, one may implement a *SHAPE-LIST* class, instances of which can hold a heterogeneous collection of *SQUARE* and *CIRCLE* objects. Given that these objects are used only as objects of type *SHAPE*, the sub-typing form of polymorphism is what is required.

However, sometimes a different form of polymorphism is required. Instead of each *LIST* class holding any kind of shape, we might instead want to instantiate *LIST* objects which hold only *CIRCLE* objects or only *SQUARE* objects; in other words, homogeneous collections. The advantage of this is that the elements of a *LIST* holding only *CIRCLE* objects can then be operated on as *CIRCLE* objects, rather than only as *SHAPE* objects. One way to do this is to create two distinct classes, *SQUARE-LIST* and *CIRCLE-LIST*, each of which implements *LIST* functionality for the appropriate element type. This is extremely wasteful, since the only differences between *SQUARE-LIST* and *CIRCLE-LIST* is in the type of element held. To avoid this wastefulness, a second kind of polymorphism, called genericity or parametric polymorphism, is supported. This form allows classes and types to be written with one or more type parameters. These type parameters are provided when an instance of the class is created. Furthermore, instances of a given generic type parameterised by different type parameters are different types, for example, *LIST[CIRCLE]* and *LIST[SQUARE]* are different types.

The basic mechanism simply allows generic parameters to be specified in class definitions, for example:

```
class LIST[T] ...
```

in which case the type parameter *T* may be used within the class definition as the type of operation parameters or results or as the type of instances variables. Objects of specialised types of *LIST* are created as follows:

```
list-of-circles : LIST[CIRCLE]
```

It is not permitted to create an instance of *LIST* without specifying the type parameter.

For container abstractions such as lists, the above form of genericity, called unconstrained genericity, is all that is required. The only operations that are applicable to entities of unconstrained generic parameter types are those operations applicable to all types (such as the assignment operation). If a generic abstraction needs to perform non-standard operations on entities of a parameter type, then that parameter type needs to be constrained to those types defining the required operation. This is called constrained genericity.

Eiffel supports a simple form of constrained genericity: type parameters may be constrained to be sub-types of a given type. Any operations applicable to the constraining type may then be used on values of the parameter type.

An alternative approach is taken in Theta [Liskov *et.al.* 1995]: generic abstractions may be constrained by **where** clauses. These clauses specify properties of type parameters, and thus acts as constraints on the types that may be associated with those parameters. This is more flexible in that the actual parameter types need not be sub-types of a given type in order to be permitted to match a formal type parameter to a given generic abstraction.

The focus of interest in this thesis is ambivalent in regard to the choice between the two. For the purpose of this thesis, the Eiffel approach is adopted for simplicity.

3.4 Class Specifications as Contracts

In a strongly-typed imperative (or functional) programming language, the signature or type of a procedure or function defines a *contract* between the implementor of the routine and any programmer whose code uses (calls) it. This contract defines the parameters the routine accepts and the results it returns. Thus, it has benefits and obligations for both parties. This type of contract is investigated in Section 3.4.1.

In strongly typed object-based languages, contracts cover objects (or classes of objects). The contract for an object covers the whole of that object's lifetime, and all operation invocations performed on it. This form of contract has advantages over procedural contracts; these are discussed in Section 3.4.2.

Type-based contracts do not incorporate semantic information about the effects of operations. Additional facilities are necessary for the specification of contracts with semantic information, this is discussed in Section 3.4.3.

Contracts are regarded as somewhat controversial, however this should not be so. The common objections and misconceptions about contracts as fundamental to successful object-based software engineering are countered in Section 3.4.4.

As in the real world, it is possible for some software contracts to be broken. Programming languages provide exception-handling mechanisms that can be used to handle this case, this is discussed in Section 3.4.5.

Polymorphism allows the substitution of a more specific object in place of a more general object to which it conforms. There is an analogous case when objects are viewed as bearers of contracts, which is discussed in Section 3.4.6.

3.4.1 Strongly-typed Sub-program Call Contracts

In a strongly typed imperative or functional programming language, there is a *contract* governing activations of each sub-program. For a given sub-program, the contract specifies that a number of actual parameters should be passed and that the types of actual parameters should match the types of the corresponding formal parameters. In return, the implementation of the sub-program is obliged to terminate, and to deliver a result of a specified type if the contract is for a function. Thus, the contract has both benefits and obligations for both parties. Namely, for the implementor of a sub-program, and for the programmer whose code calls that sub-program.

The language system is responsible for raising errors in response to violations of these obligations. These errors may be raised at compile-time, link-time or run-time. The implementor is, of course, obliged to ensure that the sub-program terminates as this cannot be checked automatically. The other checks required are type checking of procedure calls and (function) return statements, and ensuring that *when* the function returns, it returns a value. The contract is summarised in Table 5.

None of the obligations in this contract form should be cause for alarm. The majority of software engineers regard strong type checking of sub-program calls as desirable. Sub-programs that do not terminate are clearly not useful, nor are functions that return without a value. Hence, this majority of practitioners implicitly support the application of contract as described above, although it is rarely made explicit as such.

It is worth stressing the benefits that accrue to the implementor of the operation. The practical consequence of these benefits is to simplify the implementation by excluding cases outside the contract. This saves time in writing, testing and removing defects from the implementation, also it increases readability of a released implementation.

Party	Obligations	Benefits
Client Programmer	• Pass correct number of actual parameters. • Pass actual parameters with types that match corresponding formals.	• Provided with result of specified type.
Sub-program Implementor	• Ensure sub-program terminates. • Provide result value of specified type.	• May assume that actual parameters match formal parameters with respect to number and type.

Table 5 – Contractual Form for a Type-checked Procedure Call.

Party	Obligations	Benefits
Client Programmer	• ~~Pass correct number of actual parameters~~. • ~~Pass actual parameters with types that match corresponding formals~~.	• Provided with result of correct type.
Sub-program Implementor	• Ensure sub-program terminates. • Provide result value of specified type.	• ~~May assume that actual parameters match formal parameters with respect to number and type~~.

Table 6 – Contractual Form for a Sub-program Call by an Untrusted Client.

3.4.1.1 Malicious Clients

However, the contract form described above is based on the assumption that clients are only able to make sub-program calls that are type-safe. In most cases this is true. However, the sub-programs in the system-call interface of an operating system must be able to deal with malicious clients that violate this assumption. A malicious client can easily work around type-checking using typecasts (or it could be written in assembly language). By evading type checking in these ways, the client can pass mistyped values into a sub-program; clearly the implementation of this sub-program can not assume that parameters are correctly typed, an implementation that did would be vulnerable to attack by malicious clients. This attack could result in some inappropriate privilege being granted to the client, or it could simply prevent the sub-program terminating (a form of *denial-of-service* attack).

A form of contract appropriate for situations where clients cannot be trusted is shown in Table 6, this contract removes all clauses pertaining to client obligations and implementor benefits, these removed clauses are shown as ~~struck-through~~ text.

3.4.2 Strongly-typed Object Contracts

In object-based software engineering, the notion of contract extends from single operations to cover the whole of an object's lifetime. In addition to guaranteeing that each operation on an object is subject to that operation's (sub-program) contract, the contract for a class of objects ensures certain properties in regard to the object's state:

- It ensures that the state is not directly accessible to clients, in particular, the state may only be modified (indirectly) by calls of the object's methods.
- It ensures that the state is subject to a representation invariant. This is a function of the values of instance variables which is true at key points in the objects life-time, these are:
  - after the object is initialised,
  - at the start of any operation invocation, where invoked by an external client (*i.e.* excluding operation invocations within the execution of an operation on the object), and
  - at the return of any operation invocation, where invoked by an external client.

These properties result in a client obligation, three class implementor obligations and a class implementor benefit.

Party	Obligations	Benefits
Client Programmer	• Pass correct number of actual parameters *to each operation invocation.* • Pass actual parameters with types that match corresponding formals *to each operation invocation.* • *May not access object's state directly.*	• Provided with result of specified type *from each operation invocation.*
Class Implementor	• Ensure sub-program terminates. • Provide result value of specified type *for each operation invocation.* • *Must establish a representation invariant defining consistent states of the object's instance variables.* • *Must provide for the object's instance variables to be initialised to a state that satisfies the representation invariant.* • *Must ensure that any operation on the object, given that the instance variables satisfy the representation invariant, leaves that object's instance variables in a (possibly different) state that satisfies the representation invariant.*	• May assume that actual parameters match formal parameters with respect to number and type, *for each operation invocation.* • *May assume that the representation invariant is true at the beginning of execution of any of the operations.*

Table 7 – Contractual Form for Strongly typed Serial Objects.

The contract for an operation on a strongly typed object is shown in Table 7. This is based on the contract for strongly typed procedure calls (Table 5), additions are shown in *italics*. This form of contract adds four kinds of obligation and only one kind of benefit, which may seem controversial at first glance. In fact, these benefits and obligations are simply a statement of the programming discipline that is required in the effective use of modules, expressed in a slightly more rigorous form than is perhaps usual. The standard formulation is as follows:

- Certain kinds of inter-module *coupling*, in which operations in one module directly access and modify state in another, should be avoided.
- The routines in a module should exhibit high *cohesion*. This means that they should cooperate, if they access or modify shared module data (state), then they should do so according to defined rules (a representation invariant) and preserve a consistent disposition of that state.

The first of these points is the extra contractual obligation on the client, the second lists the extra obligations on the module implementor. The benefit to the module implementor is implicit in the standard notion of module, but explicit in the contract formulation.

Recall that modularity was the most influential in the software quality metrics that were identified in Chapter 2 (Section 2.3.1.1), and is universally accepted as enhancing software engineering. The strongly typed object contract form is simply a semi-formal description of the module concept. Consequently, as was the cases with the sub-program contract form, the strongly typed object contract form should alarm no one.

3.4.2.1 Limitations of the Strongly-typed Object Contract Forms

The contracts discussed above are limited in terms of expressiveness of the contractual specification. The limitations are outlined below:

- The expression of constraints on parameters and results is limited by the power of the language type-system. Furthermore, given a static type-checking regime, the type-system is itself limited to terms that are static in nature.

- The semantics of operations on objects are not included in the contractual specifications seen above. In order for contracts to serve as specifications of object behaviour, these semantic properties must be included.

The next section augments the contracting facilities with support for the expression of these kinds of information.

3.4.3 Contracts with Semantic Information

Contracts may be augmented with additional semantic information by drawing on the theory of the specification of abstract data types [Liskov and Zilles 1974, Guttag 1977, Oudshoorn 1992]. Specific language facilities are presented to provide concrete support for this semantic information in software developments.

3.4.3.1 Abstract Data Types

Formal specifications of abstract data types describe classes of data structures by the set of services (or operations) available and by the formal properties of these services. Axiomatic specifications[38] of abstract data types consist of four main sections:

- A ***types*** section, declaring the type or types introduced by the specification.
- A ***functions*** sections, declaring the signatures of services available on these types, including argument and return types.
- A ***pre-conditions*** section, restricting the domain of each of the functions.
- An ***axioms*** section, describing the semantic properties relating the functions.

The specification technique is best illustrated by example, to which end an abstract data type describing the *BANK-ACCOUNT* abstraction is developed below.

The types section for *BANK-ACCOUNT* simply defines a single type with the name *AC*. Syntactically this is expressed as:

```
types
  AC
```

In general, the types section may contain several such type declarations. Furthermore, universally quantified (generic) types may be defined. For example, one would use:

```
Stack[X]
```

to declare a generic stack type.

The signatures of the set of functions applicable to bank accounts may be expressed in the following syntax:

```
functions
  new :                                         → AC
  balance :      AC                             → decimal-2
  min-balance :  AC                             → decimal-2
  deposit :      AC ×  decimal-2                → AC
  withdraw :     AC ×  decimal-2                → AC
  transfer :     AC ×  AC × decimal-2           → AC × AC
```

The type *decimal-2* is also an abstract data type, specified elsewhere and imported into the specification of *AC*. Of these functions, only *withdraw* has a pre-condition. This is given by a Boolean expression, as in the following:

[38] There are other ways to specify abstract data types, axiomatic definition is the choice made here.

```
preconditions
  withdraw(ac : AC; amt : decimal-2)
     is (balance(ac) >= min-balance(ac) + amt)
```

This specifies that *withdraw* may only be called on accounts with sufficient balance to support the withdrawal.

Finally, there is a collection of Boolean axioms describing semantic relationships between the functions. These are as follows:

```
axioms
  ∀ ac,ac2 : AC; amt : decimal-2
     balance(ac) >= min-balance(ac)
     balance(deposit(ac,amt)) = balance(ac) + amt
     balance(withdraw(ac,amt)) = balance(ac) - amt
     min-balance(deposit(ac,amt)) = min-balance(ac)
     min-balance(withdraw(ac,amt)) = min-balance(ac)
     min-balance(first[39](transfer(ac,ac2,amt))) = min-balance(ac)
     min-balance(second[39](transfer(ac,ac2,amt))) = min-balance(ac2)
     balance(first(transfer(ac,ac2,amt))) = balance(ac) - amt
     balance(second(transfer(ac,ac2,amt))) = balance(ac2) + amt
```

These specify for all accounts, *ac*, and amounts, *amt*:

- That the balance may never fall below the minimum.
- How the *deposit* and *withdraw* functions affect the balance.
- How the minimum balance is not altered by any of the functions.
- How the *transfer* function effects the balance of both of the accounts involved.

3.4.3.2 Language Level Support

It is possible to provide much of the power of abstract data type specifications at the programming language level. The Eiffel programming language [Meyer 1992] is an existence proof. Language level specifications use Boolean assertions to provide the equivalent[40] of the preconditions and axioms sections of abstract data type specifications. There are three main kinds of assertions used in specifications.

- ***Class Invariants*** – Boolean expressions associated with a class as a whole and involving literals, constants and value-returning operations exported by the class. An object's class invariant must be true after initialisation of the object as well as before and after any remote call (*i.e.* the form o.op(p...)) of any operation exported by the class. For example, the *BANK-ACCOUNT* class might maintain as invariant the expression:

  ```
  get-balance >= get-min-balance
  ```

  specifying that, as observed by clients of the class, the balance held in an account is always greater than or equal to the minimum balance for that class.

- ***Pre-conditions*** – Boolean expressions associated with the (exported) operations of a class and involving operation parameters, literals, constants and value-returning operations exported by the class. The preconditions defined for an operation must be true prior to any call of that operation. For example, the *withdraw* operation of a *BANK-ACCOUNT* could specify the pre-condition:

---

[39] The special functions *first*() and *second*() simply select a particular element of a tuple.

[40] However, language level specifications generally do not support the universal and existential quantifiers (∀ and ∃), directly, thus are considerably diminished in power.

```
amount <= get-balance
```

This specifies that the amount to be withdrawn must be less than or equal to the current account balance.

- ***Post-conditions*** – similar to pre-conditions, these are Boolean expressions associated with the exported routines of a class. The post-conditions defined for an operation must be true when the operation completes execution. In addition to operation parameters, literals, constants and value-returning operations exported by the class, post-conditions may include expressions of the form:

```
old(E)
```

denoting the value of the expression E at the start of execution of the operation, and also may include the return value of the operation (if any). For example, the *withdraw* operation of a *BANK*-ACCOUNT could specify the post-condition:

```
get-balance = old(get-balance) - amount
```

This specifies the appropriate adjustment to the balance held in the account.

A version of the *BANK-ACCOUNT* class including assertions is given in Figure 14.

3.4.3.3 Contracts Incorporating Assertions

The addition of assertions enhances the specification by expressing semantic information about the correct functioning of the specified class. This is reflected by a strengthening of the contract associated with the class, in the ways listed below.

- Client developer obligations are strengthened: clients may only invoke an operation when the operation's pre-conditions are satisfied.
- Client developer benefits are strengthened: clients fulfilling their obligations can now expect correct behaviour of the class's operations in terms of fulfilling post-conditions and maintaining invariants.
- Class implementor obligations are strengthened: the implementation of operations must ensure that, when an operation is invoked with pre-conditions satisfied, the operation terminates with post-conditions satisfied. In addition, the class implementor must ensure that the (external) class invariant is established by the initialisation of objects of the class and is maintained by all the operations of class.
- Class implementor benefits are strengthened: the implementation of operations may assume that the class invariants and the pre-conditions for that operation are satisfied when it is called. If the pre-conditions or invariants are violated, the implementation may exhibit arbitrary behaviour, including crashing or non-termination.

The refined form of class contract incorporating semantic assertions is based on the form for strongly typed serial objects (Table 7); this form is given in Table 8.

3.4.4 Objections to Contracts Countered

The notion of contracts incorporating assertions with language support is somewhat controversial. The objections take several forms:

- objections that the obligations on client programmers are too onerous,
- objections that the obligations on class implementors are too onerous,
- objections that the benefits to client programmers are not useful,
- objections that the benefits to class implementors are not useful, and

```
class BANK-ACCOUNT
   feature {NONE}
      balance : DECIMAL
   feature
      deposit(amount : DECIMAL) is
         do
            ...
         ensure
            get-balance() = old(get-balance()) + amount
         end
      withdraw(amount : DECIMAL) is
         require
            amount <= get-balance()
         do
            ...
         ensure
            get-balance() = old(get-balance()) - amount
         end
      transfer(to : BANK-ACCOUNT,amount : decimal) is
         require
            allowed[41](withdraw(amount))
            allowed(to.deposit(amount))
         do
            ...
         ensure
            get-balance = old(get-balance) - amount
            to.get-balance = old(to.get-balance) + amount
         end
      get-balance : decimal is ...
      get-min-balance : decimal is ...
   invariant
      get-balance >= get-min-balance
end
```

Figure 14 – BANK-ACCOUNT with Assertions.

- objections to language support for assertions in contracts.

Of these, the first is by far the most widespread. Each of the objections is discussed is a separate subsection below.

The Objection that the Burden on Client Programmers is too Onerous

In a contract-based programming regime, client programmers calling a method are required to observe the pre-conditions declared by that method. This is thought by some to be too onerous a burden: instead, client programmers should have a right to expect defined behaviour from a method independent of whether they observe that method's pre-condition in their call. Usually, the behaviour defined for the case where the pre-condition is false is something along the lines of 'to exhibit harmless behaviour'.

The position resulting from this objection being sustained is common in software engineering, the resultant contractual form is shown in Table 9.

However, on closer examination, this objection is revealed as nonsense. It is clear that a method called with invalid pre-conditions cannot be relied on to perform any useful function desired by the client programmer. Given that this is the case, what is the point of the execution of such a method

[41] The ***allowed***(*c.x*(…)) operation is the logical equivalent of the pre-condition of the call *c.x*(…). This is not an Eiffel reserved word.

Party	Obligations	Benefits
Client Programmer	• Pass correct number of actual parameters. • Pass correct types of actual parameters. • Pass actual parameters in correct order. • Pass actual parameters with correct constancies. • May not access object's state directly. • *Must ensure that the pre-conditions of an operation call are true prior to performing the call.*	• Provided with specified number of results of correct type. • *Provided with an object that ensures the truth of all post-conditions specified for operations.* • *Provided with an object that ensures the truth of all class invariants at all times the object is visible to the client.*
Class Implementor	• Provide specified number of results of correct type. • Must establish a definition of consistency in the disposition of the object's state. • Must provide for the object's state to be correctly initialised to a consistent disposition. • Must ensure that the implementation of the operation, given that it acts on an object with state in a consistent disposition, leaves that object's state in a consistent (though possibly different), disposition. • *Must ensure the initialisation establishes the truth of all class invariants.* • *Must ensure that all operation invocations preserve the truth of class invariants at the termination of the invocation.* • *Must ensure that all operations produce their declared post-conditions.*	• May assume that actual parameters match formal parameters with respect to number, type, order and constancy, with the corollary that the implementor's obligations are void if this is not the case. • May assume that the object's state is in a consistent disposition at the beginning of an operation invocation. • *May assume that class invariants are true at the start of any operation invocation by an external client.* • *May assume that the pre-conditions for an operation are true at the start of any invocation of that operation by an external client.*

Table 8 – Contractual Form for Serial Abstract Data Types.

call? In the best case it will achieve nothing positive that the caller can rely on and may actually be harmful. Such calls should be eliminated from the client code. Rather than assisting this elimination, the objecting position actually retards it, by hiding instances of its occurrence.

In many cases, it is not possible for a method called with an invalid pre-condition to exhibit harmless behaviour. For example, consider the following function signature:

```
sqrt(arg : float) : float
```

Someone using this function would reasonably expect the following relationship to hold:

```
∀x : (sqrt(x) * sqrt(x)) = x
```

If *sqrt* is called with a negative argument, then it will (according to the objection), have to return a 'harmless' value. Whatever value it is chosen to return in this instance, the above relationship will be violated. In certain circumstances this violation, of which the client programmer is not made aware, can result in harmful effects. In this instance, it is not possible to provide a harmless behaviour for the method.

Next, let us consider the analogy with type checking of sub-program calls. The types of sub-program (input) parameters are simply pre-conditions in that they constrain the values that may be passed to the sub-program. There is general acceptance that a client programmer should have to call sub-programs with actual parameter types matching corresponding formal parameter types. In other words, the general view is that client programmers are required to observe typing pre-conditions. If

Party	Obligations	Benefits
Client Programmer	• Pass correct number of actual parameters. • Pass correct types of actual parameters. • Pass actual parameters in correct order. • Pass actual parameters with correct constancies. • May not access object's state directly. • Must ensure that the pre-conditions of an operation call are true prior to performing the call.	• Provided with specified number of results of correct type. • ~~Provided with an object that ensures the truth of all post-conditions specified for operations.~~ • Provided with an object that ensures the truth of all class invariants at all times the object is visible to the client. • *Given an operation is invoked with true pre-conditions, the truth of the post-conditions for that operation is ensured.*
Class Implementor	• Provide specified number of results of correct type. • Must establish a definition of consistency in the disposition of the object's state. • Must provide for the object's state to be correctly initialised to a consistent disposition. • Must ensure that the implementation of the operation, given that it acts on an object with state in a consistent disposition, leaves that object's state in consistent (though possibly different), disposition. • Must ensure the initialisation establishes truth of all class invariants. • Must ensure that all operation invocations preserve the truth of class invariants at the termination of the invocation. • ~~Must ensure that all operations produce their declared post-conditions.~~ • *Given an operation is invoked with true pre-conditions, must ensure that the operation produces its declared post-conditions.*	• May assume that actual parameters match formal parameters with respect to number, type, order and constancy, with the corollary that the implementor's obligations are void if this is not the case. • May assume that the object's state is in a consistent disposition at the beginning of an operation invocation. • May assume that class invariants are true at the start of any operation invocation by an external client. • ~~May assume that the pre-conditions for an operation are true at the start of any invocation of that operation by an external client.~~

Table 9 – Contractual Form describing Common Practice.

parameter type preconditions are required to be satisfied in this way, why not other kinds of pre-conditions also?

The most common motivation for this objection is the pursuit of robustness. This is identified in Chapter 2 as an important software engineering quality factor; as such it is worthy of attention, however it must be pursued in an appropriate form.

Robustness and correctness are separate software quality factors: correctness is concerned with the normal function of the system, robustness with the function in abnormal circumstances. Modern software engineering practice recognises the separation of these two factors in software work products. This separation is reflected in modern programming languages such as Ada and C++ in which the robustness aspect of the system's code is located in separate blocks (exception blocks and catch clauses respectively) to the code implementing the normal system function. This separation has advantages in that a reader who is concerned only with the system's function in the normal case can easily ignore the code dealing with abnormal cases[42].

The position resulting from the objection does not separate correctness and robustness. On the contrary, it necessitates conflation of the two. Given that a method must accept inputs that violate its pre-condition, the implementation of that method must check the pre-condition before proceeding to execute its normal function if the pre-condition holds. This checking, and associated recovery code, which is concerned with robustness, is not lexically distinct from the code implementing normal system function. Hence robustness and correctness aspects are conflated by the objection.

Another generally accepted principle relating to robustness is that exceptions should be handled as close to the point of their generation as possible. For example, if a particular block reads input, and, in a given instance, obtains an abnormal input value, then that block should be responsible for dealing with the abnormal value, it should not allow it to enter another part of the system. The reason for this localisation of exception handling is that the block local to the exception occurrence is the context most particular to that occurrence, and therefore has the most information relevant to its resolution. In certain circumstances, the local block will not be able to deal with the exception and will indicate such to its calling block. This escalation of exceptions is a separate issue, it does not alter the fact that the exception was in the first case presented to be handled locally, if only in part.

The position resulting from the objection does not support localisation of exception handling. Suppose some client code obtains a value and passes this value in a method call, and the value causes the pre-condition of that method call to be invalid. If the objection is allowed, this illegal value enters the method implementation, and the illegality is detected there. However, there is almost no possibility that it can be dealt with at the point of detection because there is no way for the method implementation to determine the origin of the illegal value.

The Objection that the Burden on Class Implementors is too Onerous

The burden on class implementors is that given a method is called with satisfied preconditions, that method call must return with post-conditions satisfied. A client programmer who calls a method expects it do something useful. The post-condition of a method is a declaration describing what that "something useful" is. Clearly, client programmers are entitled to expect it to be true after any valid call, otherwise they should be asking for their money back and seeking a replacement component. In fact, due to limitations in the mechanism for assertion specification, the post-condition might be only a partial specification of the behaviour offered to the client. In this case, the class implementor is obliged to ensure not only that the post-condition is true, but also that any other required behaviour (which could not be specified in the post-condition) holds as well. In this case, the burden of post-condition satisfaction on the class implementor is too light, rather than too onerous!

The Objection that the Benefits to Client Programmers are not Useful

The benefit provided to client programmers is that post-conditions will be met on method termination. This provides the client programmer using a method with (at least partial) knowledge of the effects of calling that method. It is hard to overstate the benefit of this knowledge, particularly when the client programmer and class implementor are different people, as is usually the case in large systems, or when using classes from a code library.

Integration of assertions into the source code of classes results in an additional benefit provided to *potential* client programmers evaluating an opportunity to re-use a given class. The set of pre-conditions for that class specifies the conditions under which the class may be reused (as well as used in the first place). This specification cannot become separated from the potentially reusable class.

[42] This assumes that the exception handling facility is used for handling exceptions, and other abnormal cases, rather than as a general control-flow mechanism.

Providing the class text integrates the pre-conditions, any client programmer attempting to reuse a class must confront the pre-conditions when making the decision to re-use or not.

The lack of integrated pre-conditions (and integrated semantic specifications generally), can have drastic consequences. Meyer [Jézéquel and Meyer 1997], gives a typically robust analysis of the ARIANE V fault:

1. A particular module in the ARIANE IV system had been shown to be correct in the environment of that system.
2. In the ARIANE V software development, this module was re-used rather than re-written (thereby potentially saving development time). However, the re-use was performed without reference to the specified environment of use. In fact, the document specifying this environment had become separated from the module source.
3. In the ARIANE V launch, the environment of use of the module differed from that of the ARIANE IV system. The new environment violated a pre-condition for the re-used module.
4. The consequence of the violated pre-condition was that the module failed and the rocket crashed.

Such an example clearly demonstrates the desirability of having pre-conditions integrated into the language.

Whereas pre-conditions aid client programmers in correct re-use, post-conditions and invariants can aid class implementors in correctly supporting backward compatibility. Suppose one is required to produce a new class *N*, which is to backward compatible with an existing class *O*, in addition to introducing some new functionality. The semantic criteria for backward compatibility with *O* are the assertions of that class. As is explained in Section 3.4.6, this notion of backward compatibility is subsumed an extended notion of conformance. Under the extension, the statement that "*N* conforms to *O*" implies constraints on the assertions of *N* relative to those of *O*.

The Objection that the Benefits to Class Implementors are not Useful

In concrete terms, the benefits to the class implementor have three main results. First, it is not necessary to repeat in the implementation phase work done in the previous development of the class's specification. Secondly, the ability to concentrate on the implementation of normal function, without the distraction of special cases, thus reducing the complexity of the implementation tasks. Thirdly, assertions form a useful basis for unit testing of the implemented class.

The class implementor's task is to produce a class implementation satisfying a class specification, *given* that such a specification has previously been developed. In the contract-based programming regime, declarations of assertions in the interface of an implemented class can simply be derived from the previously developed specification. This derivation is straightforward, mechanical and, with language and/or tool support may be made mostly automatic. This is clearly a significant benefit vis-a-vis the situation in which the implementation of each method checks assertions, and which requires the definition and implementation of the semantics of 'harmless behaviour' for each method.

The contract form for serial abstract data types permits the implementation of a method to assume the truth of that method's pre-condition (here including the class invariant). The implementation need not check its inputs for validity, but simply expresses its normal function. Even in the case of a single method, this reduces complexity, with consequent enhancement of readability, understandability and other software quality factors. In a chain of several nested method calls, checking inputs for validity may occur redundantly, here the benefits in reduced complexity are greater still.

Unit testing is the process of ensuring that an implementation of a class correctly implements that class's specification. It is assumed, as a basis for class implementation, that the specification is a correct description of the behaviour required of the class. Assertions provide a basis for unit testing: pre-conditions (here including the class invariant) provide a description of valid test cases, post-conditions (again including the class invariant), describe correct test case outcomes. With language/tool support for assertions, unit test generation can be substantially automated, in the absence of such support, assertions serve a useful descriptive role as the starting point for manual coding of unit tests. In either case, a substantial benefit accrues.

The Objection to Language Support for Assertions in Contracts

This position accepts assertions as useful as a software-engineering tool, but baulks at an insistence on language support [Stroustrup 1994]. There may, of course, be cases where other factors dictate the use of a programming language that happens to lack support for assertions. This is not the issue dealt with here. Rather, the aim is to show that, in the absence of such external factors, language support for assertions, such as that provided by Eiffel, is beneficial to a language's end users.

Strong typing is regarded as an important feature of most modern programming languages. In particular, type checking of sub-program calls is important in the integration of separately developed components of large software projects. As discussed above, this type checking (of parameter and result types) is simply a restricted form a pre-condition and post-condition checking. Conversely, assertion checking generalises type checking. Given this relation, a language that supports strong type checking should similarly support assertion checking to maintain consistency in language design.

Apart from this general preservation of consistency, there are several arguments in favour of language and tool support for assertions:

- In modern strongly typed languages, type checking is not differentiated from assertion checking by its being entirely static, therefore there are no grounds for separating the two on that basis.
- Only with language and/or tool support can assertions be checked statically, thus avoiding adverse impact on the performance of released software.
- Assertions serve a dual role, as specifications of the correct usage and behaviour of software modules, and for checking correctness of that usage and behaviour. Only with language support can these two roles be adequately fulfilled.
- Only with language support can robustness and correctness aspects of a software text be kept completely distinct.
- Language support for assertions can assist in locating potential opportunities for re-use.
- Language or tool support is necessary for optimal integration of assertions into the software process.

Each of these is now discussed.

A type-checking regime that is entirely static enables the type correctness of a system to be established prior to its execution. This constitutes an exhaustive proof of freedom from a class of possible errors and, as such, is much valued by anyone concerned with software quality. Furthermore, this proof is achieved at compile time. Unfortunately, purely static type systems impose unreasonable restrictions which impede the development of software, and the type systems of modern languages typically include some dynamic type checking whilst remaining mostly static in nature. Examples of non static type-checking include the Ada [ISO 1995] type system which requires dynamic checks for such types as integer ranges, and many object-oriented language type systems which require dynamic type checking to support narrowing a value of a type to a value of one of its sub-types. Hence, it can be seen that type checking is not always static, and cannot therefore be separated from assertion checking on that basis.

Just as type checking is not purely static, neither is assertion checking necessarily purely dynamic. Static checking of at least some assertions is possible (see [Detlefs 1995] for example). Static checking of assertions is several orders of magnitude more expensive in time than is static type checking. However, this time is expended at compile time, and it reduces the need for checks at run-

```
do
  if not «invariant and pre-condition» then
     raise «appropriate exception»
  else
     «implementation of method functionality»
     if not «invariant and post-condition» then
        raise «appropriate exception»
     else
        Result := «return value of method, if any»
     end
  end
end
```

Figure 15 – Method Implementation Incorporating Assertion Checking.

time, thereby improving the performance of delivered software products. Automated static checking of assertions requires assertion support from a software development tool. It is further facilitated if that tool is integrated into the language compiler and the language definition.

Assertions serve two roles in a software project. First, they document the correct behaviour and usage of software modules (classes). In an object-based language, this role requires that the assertions appear in the class interface presented to client programmers. Secondly, assertions provide the basis for automatic checking that classes behave correctly and are correctly used. This second role requires that assertions be added, as extra code, in the implementations of classes' methods. In effect[43] the implementation of a method becomes similar to that shown in Figure 15. Now, it is possible, but tedious, to manually code these checks into each method implementation. This is particularly so in languages such as C and C++[44] that allow multiple return statements, each of which must be preceded by post-condition checking code. If the manual approach is taken, then the assertion logic appears in two places, once in the class interface (presumably as comments) and once in the relevant method implementation. These two copies must be kept consistent manually, which is a continuing drain on development resources. In contrast, with language support for assertions, each assertion appears once, in the class interface, and is automatically added to the relevant method implementations by the language compiler as and when required.

Consider the situation if the manual approach to implementing the assertion checking method implementation shown in Figure 15 is followed. The result will be that pre-condition logic, which is concerned with robustness, is located in the same lexical region as the code implementing the methods functionality, which is concerned with correctness. As discussed previously, this conflation of robustness and correctness is undesirable. In contrast, with language support, the robustness logic of the pre-conditions appears in the method specification, which is lexically distinct from the implementation of the method's functionality.

---

[43] This is intended as a conceptual illustration only; in reality there are other arrangements. In particular, it may be that pre-condition checking is sometimes done at the point(s) of call to a method, rather than in the implementation of a method. This would be particularly important if static assertion checking is used, as such checking makes use of the actual parameters given in a call as a basis for inference regarding the truth of pre-conditions.

[44] C++ raises a further, potentially insurmountable problem because it supports class objects being returned by value and the ability to redefine the "copy constructor" operation for classes. Method post-conditions often contain assertions including the return value of the method. However, in methods that return, *by value*, class objects having redefined copy constructors, the action of constructing the value to be returned on the stack occurs as part of the return statement. What is needed is some way to interpose the post-condition checking between evaluation of the return statement (and construction of the return value) and the actual sub-program return, however there is no way to do this in C++. Consequently, the copy-constructor may be evaluated an additional time, thereby altering the semantics since the copy-constructor can not be guaranteed to be idempotent.

The assertions associated with a class can aid in locating classes with the potential to be re-used in a new context. Suppose the new context requires a class that implements some collection of methods, where one of the methods has to operate in an environment defined by the new context. It is possible to conceive of a library query tool that returns only classes in which the method has pre-conditions satisfactory to the new context, rather than all classes implementing that group of methods

Assertions require substantial tool support to be fully effective tools within the software process. It has been pointed out that the checking of assertions imposes a substantial run-time penalty. This penalty is surely better than defective software, and with adequate tool support, the run-time performance penalty on released software can be substantially ameliorated. As mentioned above, static checking of at least some assertions is possible in principle, however the time taken for such checks is such that it may substantially impede software development. With adequate tool support, a software project can operate in an environment where assertion checking is purely dynamic whilst software is in development, thus permitting fast compile times. When the software moves to the release stage, this same assertion checking can be performed statically (as much as is possible) thereby reducing the run-time penalty for the released software. Even if assertions cannot be proven true, it may be possible to approximate such a proof with a rigorous testing regime. Once such testing is complete, it should be possible to suppress the dynamic checks in the released system.

Limitations

It was argued above that there are no fundamental objections to object contracts in principle. However, as shown in [Meyer 1996], the contract mechanism in a programming language degrades in important practical situations. These are properly regarded as limitations in object contracts, and to some extent, limitations in the object model that are exposed by the rigour of object contracts, rather than fundamental problems with object contracts.

3.4.5 When Contracts are Broken: Exception Handling

The set of contracts between elements in a software system describes the correct, normal operation of that system. Any abnormal operation is equivalent to the violation of a (possibly implicit) contract in the system. Abnormal operation may take two forms:

- Abnormal operation deriving from defects in the engineering of the software. These result in one or more contract clauses being violated.
- Abnormal conditions detected by the environment in which the software system executes. This environment includes hardware, operating system(s), and language run-time system(s).

It is obvious that the first is a case of contract violation: an explicitly stated contract clause is not met.

The second case also involves contract violation, but here the contract may be implicit. Consider the following examples of abnormal conditions.

- Division by zero – this abnormal condition is detected by the hardware: it is a violation of the implicit pre-condition on division operations that the divisor be non-zero.
- Inadequate memory to create a new object – this is detected by the operating system and language run-time in combination: it is a violation of the implicit pre-condition on object creation that there be enough memory available to perform the creation.
- Power failure – this is detected by the operating system and hardware: it is a violation of the implicit invariant on instruction execution that the power does not fail during that execution.

If it is desired to make such contracts explicit, then it is possible to do so, for example, the contract for integer division is as follows:

```
divide(dividend,divisor : integer) : integer is
  require
    divisor !=0
```

However, it is more convenient to leave such contracts implicit.

In either case, there is a contract violated, and the system must respond in some way. The response is defined by the robustness goals of the system. Since robustness and correctness are separate, the code providing this robustness should be separate from the code providing correctness in normal function. Violation of a contract should be the trigger for the system to switch between normal and abnormal processing, and in fact should be the only mechanism to cause such a switch. There should also be a mechanism to switch back to normal processing, should the abnormal processing be able to achieve recovery. A well-designed exception handling mechanism will support these goals.

Since contract violations only occur at points of communication between modules, it is important that the mechanism for managing contract violation is integrated into the language mechanism or mechanisms that support this communication. It is therefore necessary at this point to describe a model for exception handling, in order later to be able to assess its integration within (distributed) communication mechanisms.

3.4.5.1 Exceptions and Exception Handling

Language support for exception handling, such as that of Ada [ISO 1995], attempts to separate normal and abnormal system function, with mixed success. There are three elements in the Ada mechanism.

- The ability to declare named, distinct, identifiers of exceptions.
- The ability for the execution of a routine to signal occurrence of a given exception using the **raise** operation. This transfers control to the nearest dynamically enclosing block containing a handler matching that exception, to the handler within that block.
- The **exception** clause, which is a optional component of sub-programs and certain other blocks. This clause contains one or more handlers for named exceptions, there may also be a default handler for any exceptions not explicitly named. Each handler contains recovery code specific to the particular exception being handled. This can be arbitrary Ada code, in particular, it may contain **return** and **raise** operations. After the handler completes, execution recommences as if the block declaring the exception handler had completed without an exception occurring.

Recent definitions of C++ [Stroustrup 1991, ISO 1998], provide a similar mechanism. These mechanisms achieve lexical separation of robustness and correctness aspects by segregating the robustness elements inside exception clauses. However, in practice, both fail to promote real separation of normal and abnormal function.

There are two main flaws in the Ada[45] exception handling mechanism leading to this lack of real separation:

- The ***raise*** operation can be executed by any code, at any time, for any reason. This permits its abuse as a kind of limited **goto** operation, allowing a sub-program to induce skipping of the execution of some part of its caller. This is, of course, bad in and of itself, for the reasons **goto** statements are "considered harmful" [Dijkstra 1968], although not of course to the same degree as the original **goto**, as block structure is still observed. More to the point, this form of signalling can be abused to achieve some aspect of normal function that has nothing to do with abnormal function. Hence, code that a reader might reasonably expect to deal with robustness (due to its presence in an exception clause) may, in fact, be implementing an aspect of normal function.
- There is no way for an exception handler to retry execution of the block within which an exception occurred, therefore that handler is forced either to implement normal function equivalent to that block (presumably after some abnormal processing for recovery), or to signal

[45] The C++ mechanism exhibits similar flaws.

an exception indicating inability to do so. In the first case, the exception handler must implement aspects of normal function and thereby fails to achieve separation. Unless exceptions are always allowed to propagate beyond the outermost dynamic block of a system (resulting in system shut-down), then there must always be instances of the first case.

The first problem results from the ability to use **raise** operations to perform arbitrary signalling rather than being restricted to signalling abnormalities. This problem can be corrected by eliminating the explicit **raise** operation, and only having exceptions be raised implicitly by the language run-time system when contract violation occur. One of the advantages of the Ada mechanism is that different exceptions can be used to signal different conditions and may thus be handled differently. Implicit raising of exceptions can achieve the same effect, as follows:

- In the case of violation of implicit contracts detected by the environment, a different exception can be signalled for each such violation. For example, a division by zero would result in the raising of a *division-by-zero* exception.
- For explicitly defined contracts, exception names can be associated with clauses within assertions.

An example of the second case, consider a contract containing the following assertion:

```
x and y
```

If this were re-written:

```
(not-x[46] : x) and y
```

then if the assertion that *x* is true is violated, then the exception *not-x* would be raised. If the assertion *y* were violated, then a default exception indicating assertion violation would be raised.

The C++ exception handling mechanism allows arbitrary user defined data to be passed with an exception when it is raised. It is by no means clear that this facility is desirable, but if it is, the above scheme can be extended to support it by passing the data as a kind of parameter to the assertion name.

The second problem in separating normal and abnormal function is the ability, indeed the necessity, to implement normal function in exception handlers. This is more challenging. It is not possible to eliminate the ability to perform this abuse, however it is possible (and of course desirable) to eliminate the need to do so.

A routine may either fulfil its contractual obligations, or it may fail to do so, this is a genuine bifurcation: there is no third possibility. Therefore, the occurrence of an exception during the execution of a contract leads to two possible outcomes:

- Recovery from the exception is achieved, and the contract is eventually fulfilled. This is called the retry alternative.
- Recovery is not achieved, leading to failure to fulfil the contract, which must be signalled by raising an exception (possibly the same exception) to the contract's client.

The exception handling mechanism must cater for both of these alternatives.

Whereas the explicit ***raise*** operation employed by the Ada and C++[47] mechanism was found to be counter-productive and hence was eliminated from consideration, the basic form of exception handler blocks is retained. Ada syntax for exception handling (using the ***exception*** key-word to delimit a set of exception handlers and the ***when*** key-word to specify handlers matching particular exceptions) will be adopted as much as possible.

46 The ability to name parts of precondition is not part of Eiffel.

47 The equivalent construct is called ***throw*** in C++.

3.4.5.1.1 The Retry Alternative

If the contract is eventually fulfilled, then it should be normal processing that fulfils it. This should be done by code outside of the exception handler. Rather than the exception handler executing normal processing, it should simply repair any damage resulting from the abnormal condition and then cause the normal execution of the routine in which the exception was detected[48] to resume from the start. It is this routine, after all, which is responsible for fulfilling the contract, there is no point duplicating this functionality. This is the basic semantics of the ***retry***[49] operation, due to Meyer [Meyer 1988], which we adopt here. In Eiffel, this operation takes no arguments and simply results in the routine's execution resuming from the start, but with local variables retaining their current values. The retention of local variable values is important: it permits the body of the routine to implement the required functionality in a way that is adaptive. For example, the routine body might select alternative algorithms in cases where execution is being re-attempted after recovery from an exception raised in the previous attempt. This ability to adapt can also be used to ensure that there is a bound on the number of attempts made to execute a routine for a given activation of that routine.

The availability of ***retry*** removes the necessity for exception handling code to engage in normal processing but does not prevent this abuse if the programmer so chooses. There is little that can be done here; one thing that can (and should) be done is to prohibit the use of ***return***[50] operations within exception recovery blocks.

It is very important to note that the fulfilment of a contract implies that the contracts of any method calls performed as part of that contract fulfilment were themselves fulfilled. The only way to achieve contract fulfilment when a method call made during an attempted fulfilment raises an exception is to clean up and restart the contract fulfilment.

3.4.5.1.2 The Failure Alternative

The second alternative outcome, failure of the contract and consequent escalation of the exception, is easy to support. In the case where there is no exception clause, the exception automatically escalates to the dynamically enclosing block. In the case where there is a handler, if that handler reaches the end of execution without executing a retry operation, then the exception being handled is escalated to the dynamically enclosing block.

One limitation is that the exception that is escalated is the same as the exception being handled. With the Ada mechanism, it is possible to use an explicit ***raise*** operation within the handler for some exception E_1 to raise a different exception E_2. This functionality is probably sufficiently useful to warrant the re-introduction of the ***raise*** operation in a restricted form that is permitted only within an exception handler.

3.4.5.2 Exceptions Summary

The essential points of the approach to robustness are as follows:

- Exceptions are contract violations, and only considered as contract violations. They are not used to signal arbitrary events.
- Robustness requirements are supported by a lexically separate language mechanism: exception handling. This enables separation of code implementing robustness (abnormal function) from that implementing normal function.

---

48 That is, the routine in which the exception handler is declared.

49 The detailed semantics of the retry and exception handling in Eiffel, in particular the interaction with assertions, are given by Meyer [Meyer 1997].

50 Eiffel uses assignment to the *Result* pseudo-variable and subsequent routine termination to set the return value, thereby avoiding the problem.

- Exceptions are raised implicitly, not explicitly. This partially prevents programmers from abusing the exception handling mechanism as a general signalling mechanism.
- Exception handlers have only two options, to clean up and restart execution of the associated method, or to propagate an exception (possibly a different exception) to that method's caller. This gives the contract of a method clean semantics: it succeeds completely or it fails (completely).
- Given that a method call returns normally, it is known that the contract of that method has been fulfilled completely. Complete fulfilment includes fulfilment of the contracts of any method calls made as part of the top-level fulfilment.

The final point is important, it will be revisited when object technology is extended to the concurrent and distributed case in the next chapter of the thesis.

3.4.6 Sub-typing, Substitutability and Contracts

There is an interaction between assertions and the conformance rules controlling polymorphism [Meyer 1988, Liskov and Wing 1993, Liskov and Wing 1994]. This can be summarised as follows:

- All (externally visible) invariants maintained by a class must also be maintained by all classes conforming to the type of that class.
- Any post-conditions provided by an operation in a class must also be provided by that operation in any conformant class.
- The pre-conditions of any operation for a class must logically entail the pre-conditions of that operation for any conformant class.

These observations lead to the following abstract rules for the redefinition of assertions by conformant classes:

- Conformant classes may not weaken invariants, but may strengthen them.
- Conformant classes may not weaken post-conditions, but may strengthen them.
- Conformant classes may weaken pre-conditions, but may not strengthen them.

In concrete terms, these rules imply the following logical constraints on the definition of assertions in conformant classes:

- The invariant of a conformant class must be the logical ***and*** of the invariants of all conformed-to classes and any new invariant properties added by the conformant class.
- For any operation defined in a conformed-to class, the post-condition of the operation in the conformant class must be the logical ***and*** of the post-condition of the conformed-to operation and any additional post-conditions provided by the conformant class operation.
- For any operation defined in a conformed-to class, the pre-condition of the operation in the conformant class must be the logical ***or*** of the pre-condition of the conformed-to operation and any additional pre-conditions provided by the conformant class operation.

Informally, a conformant class may strengthen invariants and post-conditions but must not weaken them, and may weaken pre-conditions but must not strengthen them[51]. An example of a class under this conformance regime is the version of *PRIVILEGED-BANK-ACCOUNT*[52] shown in Figure 16

[51] The inability to strengthen pre-conditions might be considered a serious limitation, It is, in fact, overly severe. Co-variant argument redefinition is an existence proof of valid pre-condition strengthening. The types specified in the argument list of an operation constrain the legal values of the arguments with which the operation may be invoked. Thus, these types constitute an aspect of the pre-condition for the operation. Co-variant argument redefinition of an operation further restricts the legal values of the arguments, thus strengthening this aspect of the operation's pre-condition. Given this, it is possible to apply the techniques supporting safe covariant re-definition of method arguments to the more general case of pre-condition strengthening.

```
class PRIVILEGED-BANK-ACCOUNT conforms BANK-ACCOUNT
feature {NONE}
   balance : DECIMAL
   overdraft : DECIMAL
feature
   deposit(amount : decimal) is ...
   withdraw(amount : decimal) is
      require
         amount <= get-balance
         or amount <= get-balance + get-overdraft
      do
         ...
      ensure
         get-balance = old(get-balance) - amount
      end
   transfer-to(to-accnt : BANK-ACCOUNT,amt : decimal) is ...
   get-balance : descimal is ...
   get-min-balance : decimal is ...
   get-overdraft : decimal is ...
invariant
   get-balance >= get-min-balance
end
```

Figure 16 – An Example of Conformance and Assertions.

3.5 Inheritance

Inheritance is a relationship between entities in a object-oriented system whereby the definition of a compile time entity is based on the definition of one or more other entities. Instead of the programmer describing a given entity *de novo*, that programmer may describe the entity in terms of one or more existing (base) entities, and some modifications, distinguishing the new entity from the base.

There are two kinds of inheritance relationship generally recognised in object technology. These are as follows:

- Interface inheritance, whereby an object interface type S is derived from one or more types $T_1 \ldots T_n$.
- Implementation inheritance, whereby a class D is derived from one or more classes $B_1 \ldots B_n$.

These are explained in detail in the following two subsections.

3.5.1 Interface Inheritance

Interface inheritance is a relationship between the definitions of object types. Suppose one has an object type T consisting in methods m_1, m_2 and m_3. Suppose that one wishes to add a new type S, with methods m_1, m_2, m_3 and m_4 and that the signatures of m_1, m_2 and m_3 are the same as for T. One could of course, write the definition of S in full, including the signatures of all four methods. Alternatively, one could use interface inheritance: in this case, the definition of S would include a clause inheriting from T, followed by the signature of m_4 only. This is saving in code size, which is a probable benefit to readability. Furthermore, any subsequent changes made to T are automatically reflected in S.

[52] Readers may note that the pre-conditions for the "withdraw" operation in both classes *BANK-ACCOUNT* and *PRIVILEGED-BANK-ACCOUNT* may be rewritten as:

```
amount <= get-min-balance - get-balance
```

This is sufficient to cover both definitions of the operation. However, this would eliminate the example of pre-condition weakening; for pedagogical reasons, it was thought better to continue to draw examples from the *BANK-ACCOUNT* class rather than introduce an entirely separate example solely to demonstrate assertions redefinition.

There is a relationship between interface inheritance and conformance. A type *S*, inheriting the interface of a type *T*, is automatically a candidate for conformance to *T*. In a language with structural sub-typing, interface inheritance will always imply conformance. Most languages with declared sub-typing use a single reserved word to denote both conformance and (interface) inheritance, so interface inheritance implies conformance in those languages as well. It is possible to imagine a language with declared sub-typing where interface inheritance does not imply conformance, but it is not clear how useful this extra degree of independence would be in practice.

3.5.2 Implementation Inheritance

Implementation inheritance is a relationship between the definitions of classes. Recall that a class definition contains both declarations of the state variables of objects of the class, and the implementation of methods defined by the class. Consequently, a class definition D, inheriting from class definitions $B_1 \ldots B_n$, will inherit both state variable declarations and method implementations from the union[53] of classes $B_1 \ldots B_n$,

Implementation inheritance, in being a relationship between class definitions, is a mechanism for code re-use. If the implementation of method *m*, in a class B_i, inherited by a class *D* is sufficient to implement the functionality required for *D* then B_i's method can be made available in the interface exported by *D*. In this case the implementation used is that defined in B_i, so this is an example of code re-use (method inheritance). If, on the other hand, none of the base classes provides an appropriate implementation of *m*, then such an implementation can be defined in D, this might, of course, make use of methods implemented in the base classes. This second case is termed method redefinition.

A class derived using implementation inheritance inherits the state variables defined in each of its base classes. This must be the case, given that at least one method[54] implementation is inherited from each base class, for that method implementation will require the base class state on which to operate.

3.5.3 Language Support

A given object-oriented programming language may support one or both of these kinds of inheritance. Furthermore, in languages that do support both notions, there is significant variation in the form of this support. For example, Eiffel has a single unified notion of inheritance that combines both forms. C++ supports this combined form, but also has the option of implementation inheritance without any associated interface inheritance. Java supports the combined form but has interface-only inheritance as an alternative. Finally, some languages, such as Theta, support the two forms as independent concepts.

A second degree of freedom in the provision of inheritance is the choice whether to support more than a single base entity in a given inheritance relationship. This is a language design choice to be made for each distinct form of inheritance supported. There are language examples of virtually all the conceivable possibilities.

3.5.4 The Position taken in this Thesis

No *a priori* position on inheritance is adopted in this thesis. Evaluation of the merits and drawbacks of the various approaches is mostly beyond the scope. Consequently, inheritance will receive further mention only when it has a direct impact on other topics under discussion.

---

[53] If more than one base class is allowed for implementation inheritance then there arises a complex set of language design issues concerned with the case where a given feature (method or variable) is inherited from more than one base class. These issues of conflict resolution are beyond the scope of this thesis.

[54] Implementation inheritance where all methods of a given base are redefined is usually pointless. Essentially, this would create an internal object of the base class, which is inaccessible except in the derived class methods. The proper way to support this situation would be to add an object of the (proposed) base class to the state declared in the (proposed) derived class.

3.6 Object Management Facilities

In addition to the core object model and facilities for constructing abstractions, object technology provides various facilities for the management of objects at run time. Object management facilities include:

- Automatic object lifetime management, including automatic garbage collection and support for orthogonal persistence [Atkinson *et.al.* 1983].
- Object querying capabilities.
- Abstraction of object creation.

In respect of automatic object lifetime, the position taken in this thesis is this facility is a core part of object technology in contexts where it can be implemented efficiently and entirely without burdening the programmer in any way. These contexts include sequential object systems but currently exclude distributed object systems. Since the focus of this thesis is conducted within the latter, it is not reasonable to include automatic lifetime management in this overview of core object technology.

In respect of object querying and the ability to abstract over object creation, the view taken in this thesis is that, rather than being essential parts of object technology, these facilities are useful additions to the previously identified core facilities. As such, it is not necessary to discuss these topics further.

Chapter Four

Object Technology for Concurrent and Distributed Systems

Message-passing distributed systems are concurrent systems by their very nature. This chapter discusses the introduction of concurrency into object systems within the context of a distributed system as a kind of concurrent system. Section 4.1 discusses concurrent object systems, including distributed object systems. Section 4.2 discusses additional issues relevant only in the distributed context.

4.1 Concurrent Object Systems

A traditional distinction regarding concurrency in object systems is between the active object approach versus the passive object approach. Section 4.1.1 explores this issue, arriving at the conclusion that neither model is entirely satisfactory, and describing an alternative to both called reactive objects.

Concurrency, in any form, has an impact on the object technology model developed in the previous chapter. The issues raised by this impact are discussed in Section 4.1.2.

4.1.1 Objects and Activity

The section discusses the paradigm under which concurrency should be introduced into object systems. Two possible approaches that have been adopted in the past are *active objects*, where (some) objects possess an intrinsic agent of execution, and *passive objects* where objects are acted upon by some orthogonal agent of execution.

4.1.1.1 Active Objects

In concurrent systems such as CSP [Hoare 1978, 1985] and Ada [ISO 1995], processes have an agenda of their own, a block of code which they execute. They may wait for their services to be requested by other processes from time to time, but fundamentally, they are autonomous. It has often been suggested that the object programming model is naturally concurrent. For example, in [Wegner *et.al.* 1991], Robin Milner put it concisely: "I am surprised the objects can be <u>not</u> concurrent".

Objects incorporating an *autonomous* thread of control are termed *active objects*. This is independent of whether they exist in a centralised concurrent system or in a distributed system. Active objects add a fourth fundamental object element: a thread of control, to the three existing elements (state, behaviour and identity) defined in the core object model. Active objects therefore merge the object notion with the classical concurrent systems notion of process, seen in processes in CSP and tasks in Ada.

The primary basis for this proposed unification of object and process is the identification of several similarities between the two, to wit:

- The two are similar in that both objects and processes are modules, and, in being modules, they should exhibit high cohesion and loose coupling when used appropriately.

- In addition, both objects and processes contain variables that retain their values between interactions with the object or process.
- Finally, inter-module communication between both objects and processes follows a message-passing paradigm, at least at the conceptual level, though the implementation may be based on procedure calls.

Given such similarities, one might expect the marriage of object and processes, yielding active objects, to proceed without difficulty.

In fact, there are a number of problems with active objects. Meyer [Meyer 1997], conducts an extensive critique. The main problems are given below:

- Despite superficial similarities identified above, objects and processes are discrepant in a more fundamental way. Processes have an agenda of their own, a block of statements that they execute autonomously. They may, from time to time, communicate with other processes and provide services to such, but fundamentally, they are autonomous. In contrast, objects are purely repositories of services, available to provide those services to clients on demand: they have no private agenda. Reconciling objects and processes requires complex synchronisation to take place between the process role in executing the private agenda on the one hand, and the object role in providing externally available services on the other hand. Active objects can be seen to exhibit schizophrenia of sorts.
- Active objects fail to be fully compatible with implementation inheritance because they introduce a block of code that is not a method and which therefore can not be inherited (at least by a conventional inheritance mechanism).

These problems lead to the conclusion that active objects are not an effective approach to introducing concurrency into object systems.

4.1.1.2 Passive Objects

In contrast to active objects, *passive objects* do not posses an intrinsic autonomous thread of control. Instead, instances of an orthogonal mechanism (usually called a thread) act upon the objects. As a thread executes, it calls the methods of objects. These method calls behave in exactly the same way as for sequential objects: the calling thread enters the object, executes the code of the method and then returns to the caller.

Passive objects avoid both of the problems identified for active objects. Firstly, they are pure servers, just like objects in sequential systems. Secondly, they do not introduce a block of code outside the scope of the inheritance mechanism, and are therefore fully compatible with inheritance.

In general, a passive object is accessible only to threads within the address space containing that object. In a distributed system, this implies one of the following:

- The existence of one or more distributed shared address spaces, mapped into the address spaces of the processes containing the threads.
- Threads are able to jump between address spaces on the same or different machines.

The first of these is the distributed shared memory (DSM) approach, discarded in the introduction. The second approach is the dual of DSM: ubiquitous threads instead of ubiquitous memory. This interesting alternative is supported in the Grasshopper operating system [Rosenberg *et.al.* 1996]. Unfortunately, this is not suitable in this thesis either, because it requires operating system support at each hardware node, and coordination between each of the operating systems in the system[55].

[55] In Grasshopper, each hardware node is running the same operating system, although there may be a mix of hardware architectures.

```
loop
  Wait for a request message.
  Execute request contained in message.
end loop
```

Figure 17 – The Basic Reactive Object Algorithm.

In consequence, although the passive object approach supports the object model effectively, and would be a good choice for centralised concurrent object systems, it is inappropriate for the applications addressed in this thesis, and a further alternative must be sought.

4.1.1.3 Reactive Objects

In a centralised, concurrent, passive object system, or indeed a distributed, concurrent, passive object system based on a shared global address space, all concurrency is explicit. Each client program instance (run from the command line, or run by another operating system mechanism to start a program executing) possesses a single initial thread. This thread invokes methods of objects within the process, thereby temporarily animating them. The initial thread may also create additional threads, introducing intra-client concurrency. Even without the creation of extra threads, there is inter-client concurrency since there are presumably multiple clients and they all access the same collection of objects.

However, this thesis is ultimately interested in message passing distributed object systems. Here the action of invoking a method on a (remote) object is simply the sending of a request message to that object and later reception of a reply. The client thread simply performs these actions, nothing more. In particular, the client thread does not animate the method of the remote object. What then, is the animator of the remote object? The answer is that each node in the system implicitly creates at least one thread. This thread is termed a *handler*, it executes an algorithm similar to that shown in Figure 17. There may be a more elaborate algorithm. For example, requests may be re-scheduled and executed in an order different to that in which they were received, and a given handler may be responsible for the animation of several objects. The principle, however, remains the same; these handlers are purely *reactive*. There is a crude sense in which objects in a reactive system are active. This is because the thread that executes the body of a method differs from the thread requesting the method invocation, and the thread (handler) animating the execution is in some sense intrinsic to the object exporting the method. However, at a higher level, the activity of this thread is purely to service client requests. This avoids the objections to active objects presented previously. This approach is termed *reactive objects* in this thesis, it is the approach adopted by Meyer for concurrency in Eiffel [Meyer 1997], and is the preferred model from this point on.

4.1.1.4 Caromel's Approach

Caromel's approach [Caromel 1993] has the appearance of active objects. Certain objects, those of classes inheriting from the *PROCESS* class, have a thread of control associated with them. This thread of control executes a particular method of the object (the *live* method). In other words, it is, on the surface, the active objects approach with the *live* method giving an object's body.

Caromel considers the choice between active and reactive objects in terms of a choice between active object with an explicit control and active objects with an implicit control. In Caromel's terms, the *live* method of *PROCESS* provides an explicit control. In contrast, reactive objects as described above have what Caromel would term an implicit control.

However, Caromel's approach is philosophically closer to Meyer's than might appear to be the case. The reason is that the explicit control facility (the *live* method) is intended to be used a vehicle for the construction of implicit control facilities, as is illustrated by the following quotation from a recent paper [Caromel and Vayssière 1998]:

> *"We provide two different ways for expressing synchronization policies: an explicit and an implicit one. In the explicit one, the programmer has the possibility to override the default FIFO-ordered policy by writing code for explictely (sic) managing the queue of pending calls on an object. This gives him total control over the synchronization strategy. In the implicit way, which is actually implemented using the explicit one, the programmer declares a set of properties that constrain the default FIFO ordered policy.*
> *Note that any other synchronization abstraction may be implemented using the explicit one."*

Application programmers would typically inherit an implicit control from a subclass of *PROCESS* provided by a library rather than use the explicit control facility directly by defining their own *live* method. Provision of implicit controls to application programmers is supported in several ways:

- The default version of *live*, defined in the *PROCESS* class itself provides an implicit control that is satisfactory for many purposes. In fact, this version of *live* is very similar to the prototypical reactive object handler given in Figure 17.
- The system provides a collection of reflective facilities, including the ability to reify request (and reply) messages, enabling the programmer to treat them as objects. Using this reification mechanism, subclass of *PROCESS* that redefine *live* can manipulate incoming requests and implement various scheduling policies.
- The system includes a library of *PROCESS* subclasses that implement common implicit control schemes as abstractions. Application programmer *PROCESS* subclasses can just as easily inherit from one of these abstractions as directly from the *PROCESS* class.

Caromel's approach clearly has advantages for experimentation and the development of new request scheduling mechanisms. However, there is nothing preventing application programmers from creating application specific version of *live*, in which case the objections to active objects raised by Meyer apply. Furthermore, as is generally the case with reflection based approaches, it is not clear that allowing application programmers general access to such low level details aids productivity of the software process in a production environment.

4.1.1.5 Summary

In summary, the recommended approach for supporting object-oriented concurrency is for objects to be passive reactors rather than active actors. Furthermore, for this regime to exist by construction of the language design, as in Meyer's approach (sometimes called SCOOP [Meyer 1997]), rather than only by convention, as in the preferred programming style for Eiffel //.

4.1.2 The Impact of Concurrency on Object Technology

The introduction of concurrency into an object-oriented programming language prompts several questions:

- How much concurrency should be permitted within an object?
- How do objects that are executing asynchronously synchronise when required?
- Given that objects receive concurrent requests, how is the handling of these requests scheduled?
- If a client object can have several requests in progress at a given time, how are the replies to these requests scheduled when they arrive?

Each of these questions is answered below.

4.1.3 Concurrency within Objects

The two forms of concurrency that can exist within an object system are as follows:

- *Inter-object concurrency* – there may be several agents of execution in the system, but no object is subject to concurrent access by more than one agent.
- *Intra-object concurrency* – a given object may be subject to concurrent access from two or more agents of execution.

The second form should be considered to include the first, therefore the choice is not between the two forms but is whether to support intra-object concurrency in addition to inter-object concurrency. This choice is discussed in Section 4.1.3.3.

As discussed in the previous chapter, the notion of class invariant is very useful as a basis for reasoning about software. Class implementors can rely upon the class invariant holding at the beginning of execution of each method, providing each method ensures that the class invariant holds at the end of its execution.

When concurrency is introduced, there is a potential threat to this simple basis for reasoning. Consider an arbitrary collection of method calls, each of which temporarily invalidates the invariant during execution, executing against a single object. Only the first of these to commence will find the invariant true; for the remainder, it will not hold. As a result, each of these methods (and in general all methods) need to be written without the benefit of assuming the invariant to be true. This is unacceptable (at least within the software engineering approach advanced by this thesis). There are three possible resolutions of this problem, each allowing progressively more concurrency:

- Prohibit any concurrency within objects. All requests that arrive while a method call is in progress will have to wait.
- Allow concurrency subject to a readers/writers protocol. Methods are allowed to run concurrently if they do not modify the object's state, and hence never invalidate the invariant; all other methods must run with exclusion.
- Treat method calls as transactions. This allows arbitrary concurrency but with each method call appearing to run in isolation, and with the possibility that method calls need to be rolled-back and restarted.

Of these, the third usually is too heavyweight to implement efficiently, the exception being the case where the objects reside within an object-oriented database: transactions are *de rigeur* in such environments.

Java [Arnold and Gosling 1997] allows the programmer of a class to specify (via the ***synchronized*** reserved word) that method calls should acquire a per-object lock before proceeding. Such Java classes follow the first alternative identified above. A similar locking approach could be used in conjunction with compiler or programmer identification of non-mutating methods to provide the second alternative. In an environment such as Java, where threads can be explicitly created, and hence any object may become subject to concurrent access, the use of locking is unavoidable. Unfortunately, this has serious implications for performance: the call overhead for a ***synchronized*** method call in Java is an order of magnitude larger than is the overhead for a method call which does not need to acquire the object lock.

4.1.3.1 Eiffel //

The reactive objects approach abandons explicit creation of threads and, hence, the need for locking on all objects. Eiffel // provides the first of the three alternatives identified above. An Eiffel // system consists of two kinds of objects: reactive, top-level objects (those descended from the class *PROCESS*), and other, passive, objects. The top-level objects have a single thread of control associated with them, this thread is responsible for servicing any requests on the object. Top level

1. If the source of an attachment (assignment or argument passing) is separate, its target entity must be separate too.
2. If the actual argument of a separate call is of a reference[56] type, the corresponding formal argument must be declared separate.
3. If the source of an attachment is the result of a separate call to a function returning a reference type, the target must be declared as separate.
4. If the actual argument of a separate call is of an expanded type, its base class may not include, directly or indirectly, any attribute of a reference type.

Figure 18 – Meyer's Separateness Consistency Rules.

objects may contain references to passive objects and to other active objects, however, a given passive object may be referenced only within a single top-level object. In any communication between top-level objects involving passive object values, copies of the closures of those passive objects are passed rather than references to the objects themselves. Therefore, Eiffel // passive objects are never subject to concurrent access, and need not have a per-object lock. In Eiffel //, there is a rigid system structure in which a group of top-level objects (those of classes derived from the *PROCESS* class) entirely contain instances of other classes. Such instances are effectively private to a given *PROCESS* object.

4.1.3.2 Concurrency in Eiffel (SCOOP)

In SCOOP (the concurrency facilities for Eiffel), concurrent execution is performed on handlers (sometimes also called virtual processors), each of which is a single threaded agent of execution. Each object has only one handler, and thereby SCOOP enforces a regime in which there is at most one thread of execution within a given object. The ***separate*** reserved word is an attribute applied to object references to indicate that the object referred to is executed on a different handler than the object holding such a reference. A given object may be referred to via both separate references (from objects with different handlers) and non-separate references (from other objects with same handler as the object).

There may be several objects on a given handler referenced via separate references from objects on other handlers: there is no single top-level object as is the case with Eiffel //, so the SCOOP facility offers increased flexibility. Furthermore, the ***separate*** reserved word is an attribute of object references. Therefore, it is possible to create both separate and non-separate objects of a given class. This is another increase in flexibility *vis-a-vis* Eiffel //, in which all instances of classes descended from the *PROCESS* class are the equivalent of separate objects. For convenience, SCOOP does allow the ***separate*** reserved word to be applied to classes, in which case all instances of that class are automatically separate objects.

Like Eiffel //, SCOOP places restrictions on references that cross concurrency boundaries (references to objects with different handlers). However, SCOOP is much more flexible than Eiffel //. In SCOOP, the essential requirement is to ensure that all references to an object from objects with different handlers are separate references. Meyer refers to a non-separate reference to a foreign (separate) object as a *traitor*. The exclusion of such traitors is enforced by the semantic rules shown in Figure 18. Notice that SCOOP does not seek to prevent a separate reference becoming attached to a non-separate entity.

[56] Eiffel class objects can be of either reference or expanded types. Passing an argument in the former category simply passes a reference to an existing entity (on the local processor), a reference passed in a separate call establishes a inter-processor reference which must therefore by designated as separate. Expanded types (including user defined expanded types and basic types such as integers and floating point numbers) are passed by value. Consequently, the destination

(continued)

In Eiffel // inter-handler references are restricted to refer only to the top level (PROCESS) object on each handler[57]. Closure copying is used to pass non-PROCESS objects between handlers. Eiffel (with SCOOP) is more flexible: any (reference type) object may be referenced from a non-local processor.

SCOOP can also approximate the Eiffel // semantics via a facility to allow copying of object graph closures between processors. The *GENERAL* class has the following function:

```
deep-import(other : separate GENERAL) : GENERAL is ...
```

This makes a (non-separate) copy of the closure of *other* as it is on the processor containing it. However, there are two things to note:

- If the original closure of *other* contained separate references, the copy also will contain those separate references. The *deep-import*() function removes only one level of separateness[58].
- Transmitting a deep copy of an object graph on a client to a server is a two-stage process. First, the client sends the server a separate reference to the root of the graph. Second, the server imports the closure of the reference it receives. It is possible for the client to mutate the object graph after passing the reference to the server, but prior to the server commencing importation, this may lead to confusing results. In addition, the two-phase process takes at least three messages to complete, this may be considered inefficient in wide area networks.

  Perhaps a better alternative is to allow the programmer to specify that an object graph is to be closure copied as part of argument passing to a separate call. The server then obtains a non-separate reference to a copy of the object graph as an argument. This process is atomic, and requires only one message transmission. Notice that this proposed alternative semantics is the same as the semantics provided by Eiffel // in transmitting non-*PROCESS* objects between *PROCESS* objects.

  An extreme form of the previous idea is for the language implicitly to apply a closure copy operation automatically. This would occur where an actual argument of a separate call is of a reference type and the corresponding formal argument is not declared separate. In SCOOP, such a call would be illegal: it contravenes rule 2 in Figure 18. Supporting such an extension would mean calls exhibited radically different semantics depending on whether certain actual arguments were separate objects. Whilst technically this semantic distinction is reflected in a syntactic distinction, this syntactic distinction is not at the point of the call. Therefore it is not necessarily clear to the reader of the call and, hence, confusion could quite possibly arise. In this light, it seems more sensible to revert to the previous idea wherein an explicit indication was made at the point of the call.

  Apart from the problems with passing object graph closures in inter-processor calls, the SCOOP approach, in providing flexible regions of concurrency via the ***separate*** reserved word, appears to be most appropriate for supporting object-oriented concurrency. In addition to this basic notion of separateness, SCOOP includes other concurrency facilities; these are discussed later in this chapter.

4.1.3.3 Additional Concurrency for Reactive Objects

In both the Eiffel // and SCOOP reactive approaches, the implicit nature of the agents of execution ensured that an object's state is subject to access by at most one thread at any given time. The SCOOP notion of handler is a single thread of execution. Only the handler can directly access the state of the

processor of a separate call gets a copy of the argument, and no inter-processor reference is created. Hence, there is no need for separate in this case (see also separateness consistency rule 3).

[57] The handler concept is from SCOOP not from Eiffel //, it is here applied to Eiffel // for uniformity in nomenclature.

[58] A *deepest-import* function that made embedded references non-separate as well would be a useful addition.

objects it handles, and an object has only one handler. Therefore, objects are never subject to direct concurrent access, only to indirect concurrent access, involving inter-handler (separate) calls. Eiffel // works in a similar fashion, with the thread associated with a *PROCESS* object substituting for the handler entity.

These approaches can be generalised to support limited concurrency within handlers by allowing each handler at any given time to execute either of the following:

- A single thread of execution, which might mutate one or more of the objects resident in the handler.
- Multiple threads of execution, each of which is guaranteed not to mutate any of the objects resident in the handler. These are read-only threads.

Threads are started by a handler in response to separate calls on resident objects originating with external objects. The handler would start a read-only thread for an incoming separate call if it was guaranteed not to mutate any local objects, and a start mutating thread otherwise. The guarantee of non-mutation may be expressed in source code by marking appropriate methods with the ***readonly*** reserved word. The body of a method marked ***readonly*** would be prohibited from assigning to the state of any object, and from making non ***readonly*** method calls.

4.1.4 Synchronisation of Asynchronous Agents of Execution

The introduction of concurrency to an object system implies that some elements in that system will execute asynchronously by default. At times, communication between those elements is necessary. To effect safe communication between these elements requires synchronisation.

Before proceeding to investigate the alternatives for synchronisation, it is important to realise that communication and synchronisation are inextricably linked. Communication without synchronisation is unsafe. Synchronisation intrinsically involves communication: even 'pure' synchronisation not involving the exchange of data values communicates timing information. It is desirable that this linkage at the conceptual level be reflected at the language level. Hence, a language designer should aim for a (data value) communication mechanism which has associated synchronisation semantics, rather than separate language features for data value communication and synchronisation. This excludes 'pure' synchronisation constructs such as semaphores and critical sections from consideration. Instead, this thesis concentrates on synchronisation semantics associated with communication constructs. In the case of object technology, the communication construct is the method call operation. It is worth noting that since the advent of the first version of CSP [Hoare 1978], most synchronisation facilities in concurrent languages (independent of whether the language is object-based or not) have followed the communication based approach.

There remains the question of precisely how communication and synchronisation are to be associated. There are in fact three parts to this question:

- How does an object reserve exclusive access to one or more separate objects for an extended period?
- How does an object that is subject to concurrent incoming requests, schedule those requests?
- How does an object, having made a request to a concurrently executing object, schedule the reply to that request?

These questions are the subjects of the next three subsections.

4.1.5 Object Reservation

Most concurrent programs contain critical sections. These are sections of code within which a thread must obtain and maintain exclusive access to one or more shared objects. Consider the example shown in Figure 19 (adapted from that given by Meyer [Meyer 1997]). In a sequential program, the

```
q : separate QUEUE
...
a := q.head  -- Get item at head of queue.
… Other instructions, unrelated to q …
q.remove        -- Remove item from head of queue.
```

Figure 19 – Access Requiring a Critical Section.

```
head : QUEUE-ITEM is
  require
     not empty
  do
     … Result := head element.
  end
```

Figure 20 – A Separate Method with a Pre-condition.

item retrieved by the call to *head* will be removed by the call to *remove*. In a concurrent program, this may not be the case; another concurrent client of *q* may have modified it between the two calls[59].

What is needed here is for the client executing the code shown above to have the ability to reserve *q* for its own exclusive access whilst it executes instructions between the call to *head* and the call to *remove*. The section of code in which the client has exclusive access is called a critical section.

To complicate matters, critical sections may be have conditional entry requirements. In the example just given, the client is only interested in entering the critical section when the queue is not empty. Even if no other client is currently using the queue, the client is unable to use it if it is empty.

The *head* method of class *QUEUE* is shown in Figure 20. If the client did try to call *head* on a queue that was empty, it would receive, quite properly, an exception for having violated the contract associated with the queue. Instead of this undesirable outcome, the client ought to wait until the queue is not empty before entering the critical section. Therefore, the requirement for critical sections includes an optional initial condition as well as exclusive access to the objects reserved in the section.

SCOOP proffers a simple approach to the provision of critical sections with entry requirements. In SCOOP, critical sections are methods. The scope of a critical section is a method body. Of course, not all methods are critical sections! Only those methods taking one or more separate formal arguments act as critical sections. Prior to execution of the body of such a method, exclusive access to each (non-null) separate parameter is obtained. This may induce a wait for some or all of these separate objects to become available. Exclusive access, once gained, is not released until the end of execution of the method body. If the SCOOP approach is used, method decomposition needs to be informed by the need to avoid deadlock resulting from implicit reservations.

4.1.5.1 Conditional Reservation and the Precondition Paradox

SCOOP adds conditionality to critical sections through a modification of the semantics of pre-conditions involving separate objects. Pre-conditions normally have exception semantics: a method pre-condition that is not true generates an exception to the caller. For the case where a pre-condition involves a separate object however, that pre-condition's value is changing asynchronously with respect to the client. Such pre-conditions therefore are defined to have wait semantics in that they delay the execution of the method until they become true (reservation of the separate arguments can be considered an implicit wait pre-condition).

Meyer refers to the need for this dual semantics of preconditions as the *precondition paradox*. The SCOOP implementation of the queue access example is shown in Figure 21. SCOOP imposes the

[59] This client might be called an *interloper*, to go with the traitor previously identified.

```
access(q : separate QUEUE) is
  require
    not q.empty
  do
    ...
    a := q.head   -- Get item at head of queue.
    ...           -- Other instructions, unrelated to q.
    q.remove      -- Remove item from head of queue.
    ...
  end
```

Figure 21 – A Critical Section in SCOOP.

requirement that any call to a method (including calls made as part of assertion evaluation) of a separate object must occur in a method for which that object is a separate parameter. Hence, the target object is reserved in a critical section. The aim of this rule is to prevent programmers from introducing the kinds of defects that can occur when asynchronous objects are used outside of critical sections. The motivation is that these kinds of defects are so costly to diagnose that it is best for the language to prevent them rather than rely on the programmer to do so. There is more detail involved in the semantics of reservation, for this see [Meyer 1997], however it is not necessary to recount these details in this thesis.

4.1.5.2 Critique of Object Reservation in SCOOP

The SCOOP support for critical sections is certainly elegant, and apparently simple. The question is whether it is in fact too simple, simplistic and inflexible. A number of objections might be made against this aspect of SCOOP, including the following:

- The programmer is forced to introduce extra methods to act as critical sections.
- The system reserves all separate formal parameters of a method, even those that are not used as the target of a method call in that method.
- A separate object must be a formal parameter of any method wherein it is used. Consequently, it will be reserved for the entire direction of the method execution, *even if this is not necessary to ensure correct method operation.*

The first of these is specious. Critical sections contain a collection of calls to separate objects, such that this collection of calls operates correctly only when exclusive access to those separate objects in maintained through the duration of the calls. Once correctly implemented, such a collection of calls is in fact a valuable abstraction, which might be reused by other clients needing the same facility. As such, the critical section ought to be made a method on methodological grounds, even if it were not forced to be so (by the language).

In contrast to the first objection, the second and third objections are more serious. They are discussed with the aid of a detailed example in the remainder of this section

Consider a class *PRINT-AGENT* representing a printing agent, that is a class that accepts a list of print requests and forwards them to be printed by the printer (represented by the class *PRINTER*). As part of the forwarding, the agent might add extra information such as a customised header page. Each *PRINT-AGENT* may be used with several *PRINTER* objects.

A first attempt at an implementation of *PRINT-AGENT* is shown in Figure 22. Unfortunately, this implementation contains two serious defects. The body of *print* is not conceptually a critical section, and the print agent does not require exclusive access to the printer for the entirety of this access, but SCOOP nevertheless provides exclusive access. This of course means that other clients competing for

```
class PRINT-AGENT-1 is
feature {NONE}
  print-job(dest : separate PRINTER, job : PRINT-JOB) is
     require
        job.paper-requirement <= dest.paper
     do
        dest.print(job)
     end
feature
  print(dest : separate PRINTER, jobs : PRINT-LIST) is
     do
        for j in jobs loop
           print-job(dest,j)
        end
     end
  … Methods to set options etc …
end
```

Figure 22 – Initial version of PRINT-AGENT.

the printer will be locked out until all of the print jobs in the *PRINT-LIST* have been printed. This exclusion is grossly unfair, given that it is unnecessary.

The second defect is worse, the pre-condition of *print-job* has wait semantics (because *dest* is a separate formal parameter). However, if the pre-condition is false when *print-job* is called, there is no point waiting. This is because the client already has the printer reserved (as a result of the call to *print*), and there is therefore no possibility of this pre-condition becoming true since no other client can access the printer.

The second version of the print agent, shown in Figure 23, removes these defects, but does so at great expense. Three extra methods have been added. Worse, an item (*current-printer*) has been added to the state of the class for entirely artificial reasons. Worse still, the protocol for requesting the services of the print agent is now two methods (*set-printer* and *print*) called in that order, rather than a single method as previously. The simplicity of the print agent has been contaminated by various SCOOP requirements and restrictions.

Now, suppose one agrees with the second objection to SCOOP, and consequently modifies the semantics so that a method with a separate formal parameter reserves that parameter only if it is used as the target of a separate method call within the method body. With these semantics, the implementation given in Figure 22 will no longer be subject to the unfairness and infinite wait defects.

Unfortunately, the changed semantics has a drawback as well: it is no longer possible to tell which objects are reserved in a critical section. Previously, one could simply inspect the argument list for separate parameters. Any such parameters were then known to be reserved. With the new semantics, one only knows that separate formal parameters might be reserved. To be sure, it is necessary to examine the method body and associated assertions.

In response to this problem, it has been proposed to add an extra ***hold*** clause [Jalloul and Potter 1991, Jalloul 1994, Schmidt and Chen 1995] to methods enabling explicit specification of the separate objects to be reserved. This clause simply lists the objects to be reserved, the definition of the *print-job* method then becomes that shown in Figure 24. Separate pre-conditions of methods with a hold clause continue to have wait semantics, and it is an error to use a separate object in a method call within a pre-condition unless that separate object appears in the ***hold*** list. Also, note that with hold lists, separate objects do not need to appear in the formal parameter list in order to be useable.

```
class PRINT-AGENT-2
feature {NONE}
  current-printer : separate PRINTER
  print-job(dest : separate PRINTER, job : PRINT-JOB) is
     require
        job.paper-requirement <= dest.paper
     do
        dest.print(job)
     end
feature
  has-current-printer : BOOLEAN is
     do
        Result := current-printer /= null
     end
  set-printer(dest : separate PRINTER) is
     do
        current-printer := dest
     end
  print(jobs : PRINT-LIST) is
     require
        has-current-printer
     do
        for j in jobs loop
           print-job(current-printer,j)
        end
     end
end
```

Figure 23 – A Cumbersome version of PRINT-AGENT.

```
print-job(dest : separate PRINTER, job : PRINT-JOB) is
  hold
     dest
  require
     job.paper-requirement <= dest.paper
  do
     dest.print(job)
  end
```

*Figure 24 – Definition of the print-job Method with a **hold** Clause.*

4.1.5.3 Server Side Queuing Semantics for Preconditions

Even with both modifications to the semantics (***hold*** lists and the removal of the requirement that separate objects be formal parameters), the mechanism is still not entirely satisfactory for the *PRINTER* example. The method *print-job* is implemented as a critical section, and the client waits for the printer to have sufficient paper before it executes the body of the method and sends off the print job. With the semantics of SCOOP, the client must wait to ensure that it does not violate the precondition of the *print* method of class *PRINTER* (shown in Figure 25).

However, in line with the third objection to SCOOP, there is actually no reason why the client should wait. It would be just as satisfactory for execution of the *print* method of class *PRINTER* to be delayed *at the server (PRINTER) end* until its pre-condition becomes true (this is discussed further in the next section). The client would simply send the request off and then would be free to continue with other processing.

The client programmer can take advantage of this by re-implementing the *print-job* method as shown in Figure 26. Notice that there is no ***hold*** list for this method. The semantic constraints are

```
class PRINTER
  paper : INTEGER is do ... end

  print(job : PRINT-JOB) is
     require
        job.paper-requirement <= paper
     do
        ... Print the job ...
     end
end
```

Figure 25 – The PRINTER Class.

```
print-job(dest : separate PRINTER, job : PRINT-JOB) is
  d
     dest.print(job)
  end
```

Figure 26 – Definition of the print-job Method using Server-side Queuing.

further modified so that, within the method body[60], calls can be made to separate objects that do not appear in the hold list. The dynamic semantics of such calls is for the client to continue immediately after submitting the call. The call is simply queued to be executed by the server. It may be that the execution of the call does not commence until some later time. It might also be that the client needs a result of a call. The semantics of reply scheduling are discussed in Section 4.1.7.

4.1.5.4 Summary of Precondition Semantics

It is clear that preconditions for calls to asynchronous objects cannot continue to have the pure correctness semantics of calls to synchronous objects. Instead, the semantics of such preconditions must be altered to include an element of synchronisation. Two solutions have been proposed:

- As proposed by Meyer, that clients be delayed until specified preconditions relating to server they intend to call are true, and when permitted to proceed, they be granted exclusive access to those servers.
- That each call be queued at the server and not executed until the preconditions of that calls are true.

These are complementary and ideally one would support both. However, the first of these may not be suitable for middleware in linguistically heterogeneous systems, in that it requires significant support from the client implementation language (syntactic distinction of separate objects, *etc.*).

4.1.6 Request Scheduling

Given that (at least some) objects are accessible to more than one thread of control, a given object can receive several requests concurrently. The task of managing concurrent requests is called *request scheduling* by Papathomas [Papathomas 1992].

The simplest approach to request scheduling is to execute incoming requests "in the order they arrive". This ordering may be established by placing all incoming requests on the tail of a queue which is protected by a critical section. Requests are then removed from the head of the queue for execution.

[60] Calls made in pre-conditions can still only use those separate objects declared in the hold list. It is difficult to see any cases where a call to a separate object that is not reserved within a pre-condition would not be an error.

Unfortunately, this approach does not work. Consider two clients, each executing a series of requests such as the following to a shared object called S:

```
S.lock ... S.unlock
```

where lock and unlock have the expected semantics of obtaining and releasing exclusive access to the object (they provide a primitive form of critical section). In isolation, each client uses S correctly, however, in concurrent combination, the two clients may present the following trace of requests to the object:

```
S.lock
...
S.lock
...
S.unlock
...
S.unlock
```

This trace is illegal in that it attempts to lock an already locked object. To avoid this, there needs to be a mechanism to re-schedule concurrent incoming requests so that they follow a correct trace. Programmers need to be able to control this rescheduling in some way.

Two main approaches to controlling request scheduling are identified in [Matsuoka, Taura and Yonezawa 1993]. These are as follows:

- Accept-set specifications.
- Guarded methods.

These will be described below. A persistent problem in the development of request scheduling facilities is the so called 'inheritance anomaly', this is also described.

4.1.6.1 Accept-Set Specifications

An accept-set is a collection of methods that may execute at some point in an object's lifetime. As part of a methods execution, the current accept-set for the object might be altered. Accept-set language facilities consist of two parts:

- At the class level, a collection of definitions of ***accept-sets*** for that class. Each such definition consists of the name of the set and the set of methods to be enabled.
- Within the implementation of methods, a special operation (called **become** by Matsuoka), which changes the object's current accept set.

The request scheduling information is consequently partly at the class specification level and partly at the method implementation level.

4.1.6.2 Guarded Methods

In the guarded methods approach, there is a Boolean predicate associated with the specification of each method. Only when this predicate is true may the method be executed. The predicates themselves are evaluated under mutual exclusion by the language run-time support system, and the predicate associated with a given method might be evaluated several times before a given invocation is permitted to proceed. This approach has the advantage that all of the request scheduling information is in one location.

Method pre-conditions, as used in sequential object technology, provide the basis for method guards. Consider the *LOCK* class, shown in Figure 27, initially in a sequential environment. In a sequential environment, a client that attempts to execute the lock method on a locked object will generate an error: the method call is in violation of the contract.

Now consider the use of *LOCK* in a concurrent environment. If a request to the *lock* method arrives whilst the object is locked, then method execution should simply be delayed until the object is no

```
class LOCK
feature {NONE}
  lock-state : BOOLEAN
feature
  locked : BOOLEAN is
     do
        Result := lock-state
     end
  create is
     do
        lock-state := false
     end
  lock is
     require
        not locked
     do
        lock-state := true
     end
  unlock is
     require
        locked
     do
        lock-state := false
     end
end
```

Figure 27 – A LOCK class using pre-conditions to specify concurrency control.

longer locked. In other words, the pre-condition of the *lock* method, *exactly as written for the sequential LOCK class*, acts as a method guard for concurrently accessible instances of the *LOCK* class.

Not all pre-conditions should act as method guards, however. For example, there is no point delaying a call to a square root method with a negative argument. Such a call should fail immediately. The distinction is determined by whether a pre-condition involves values of instance variables or calls of methods of objects that execute concurrently with the delay of the call. Such pre-conditions have *delay* semantics, others retain the failure semantics of the sequential case. The most obvious example of delay pre-conditions are those involving the target object itself (as is the case with the *LOCK* class), but there may be others, for example, a third party concurrent object may be referenced in a pre-condition.

Since guarded methods and contract-based programming are so clearly compatible, it would seem that this approach, rather than accept sets, is the appropriate construct for request-scheduling in concurrent object-oriented programming. However, before making a firm commitment to this choice, it is necessary to examine an issue that has plagued the integration of request scheduling with object-oriented technology: the *inheritance anomaly*.

4.1.6.3 The Inheritance Anomaly

The inheritance anomaly is simply the identification of a set of incompatibilities between request scheduling mechanisms and (implementation) inheritance.

Matsuoka *et. al.* [Matsuoka *et. al.* 1993], give three distinct cases in which the incompatibilities arise:

- State partitioning.
- History-only sensitivity of acceptable states.

```
class BUFFER-AS
accept sets
   empty : put
   partial : put,get
   full : get
feature {NONE}  -- Private features
   in : INTEGER
   out : INTEGER
   size : INTEGER
   contents : ARRAY<ITEM>
feature              -- Public features
   create(s : INTEGER) is
      do
         size := s in := 0 out := 0
         contents :=
            new ARRAY<ITEM>(size)
         become empty
      end
   get : BUFFER-ITEM is
      do
         out := out + 1
         if in = out then
            become empty
         else
            become partial
         end
         Result :=
            contents[out mod size]
      end
   put(i : BUFFER-ITEM) is
      do
         in := in + 1
         if in = out + size then
            become full
         else
            become partial
         end
         contents[in mod size] := i
      end
end
```

```
class BUFFER-AS2 inherit BUFFER-AS
accept sets
   one : put,get
   partial : put,get,get2
   full : get,get2
feature
   get : BUFFER-ITEM is
      do
         out := out + 1
         if in = out then
            become empty
         elseif in = out + 1 then
            become one
         else
            become partial
         end
         Result :=
            contents[out mod size]
      end
   get2 : BUFFER-ITEM-PAIR is
      do
         out := out + 2
         if in = out then
            become empty
         elseif in = out + 1 then
            become one
         else
            become partial
         end
         Result := new BUFFER-ITEM-PAIR(
            contents[(out-1) mod size],
            contents[out mod size])
      end
   put(i : BUFFER-ITEM) is
      do
         in := in + 1
         if in = out + size then
            become full
         elseif in = out + 1 then
            become one
         else
            become partial
         end
         contents[in mod size] := i
      end
end
```

Figure 28 – State Partitioning Inheritance Anomaly.

- Modification of acceptable states.

These are explored using a collection of examples adapted from those presented by Matsuoka *et. al.*

4.1.6.3.1 Inheritance Anomaly: State Partitioning

The state partitioning case occurs only with the use of accept set request scheduling mechanisms. Consider the *BUFFER-AS* class defined in Figure 28. Note the additional constructs to support accept sets: the ***accept sets*** region and the ***become*** operation.

```
class BUFFER-G
feature {NONE}
   in : INTEGER
   out : INTEGER
   size : INTEGER
   contents : ARRAY<ITEM>
feature
   create(s : INTEGER) is
      do
         size := s in := 0 out := 0
         contents :=
            new ARRAY<ITEM>(size)
      end
   empty : BOOLEAN is
      do
         Result := in = out
      end
   full : BOOLEAN is
      do
         Result := in = out+size
      end
   get : BUFFER-ITEM is
      require
         not empty
      do
         out := out + 1
         Result :=
            contents[out mod size]
      end
   put(i : BUFFER-ITEM) is
      require
         not full
      do
         in := in + 1
         contents[in mod size] := i
      end
end
```

```
class BUFFER-G2 inherit BUFFER-G
feature
   partial : BOOLEAN is
      do
         Result := in > out+1
      end
   get2() : BUFFER-ITEM-PAIR is
      require
         partial
      do
         Result := new BUFFER-ITEM-PAIR(
            contents[(out-1) mod size],
            contents[out mod size])
      end
end
```

Figure 29 – State Partitioning Inheritance Anomaly: Solution with Guarded Methods.

Suppose now that it is desired to develop an extension of the *BUFFER-AS* class, which supports an operation to get two items out of the buffer at once. This class, also given in Figure 28, should be related by inheritance to the *BUFFER-AS* class. Very little of the class *BUFFER-AS* may in fact be re-used in *BUFFER-AS2*: inheritance is almost completely impotent here.

The problem is that partitioning of the *partial* state/set in *BUFFER-AS* into the *partial* and *one* states/sets in *BUFFER-AS2* requires that the implementation of both *put* and *get* be redefined to account for the new state/set. This is unavoidable if accept sets are used to control request scheduling and is a strong argument against them.

The BUFFER abstraction can be rewritten using method guards [Ferenczi 1995] (pre-conditions). The resulting classes, *BUFFER-G* and *BUFFER-G2*, are shown in Figure 29. Here it can be seen that method guards do not suffer from this form of inheritance anomaly.

4.1.6.3.2 Inheritance Anomaly: History-only Sensitivity

The second kind of inheritance anomaly identified by Matsuoka *et. al.* [Matsuoka *et. al.* 1993] is "history-only sensitivity of acceptable states". This occurs when the availability of method to be executed is dependent, in whole or part, on the trace of the methods executed on the object in the past.

```
class BUFFER-G
feature {NONE}
   in : INTEGER
   out : INTEGER
   size : INTEGER
   contents : ARRAY<ITEM>
feature
   create(s : INTEGER) is
      do
         size := s in := 0 out := 0
         contents :=
            new ARRAY<ITEM>(size)
      end
   empty : BOOLEAN is
      do
         Result := in = out
      end
   full : BOOLEAN is
      do
         Result := in = out+size
      end
   get : BUFFER-ITEM is
      require
         not empty
      do
         out := out + 1
         Result :=
            contents[out mod size]
      end
   put(i : BUFFER-ITEM) is
      require
         not full
      do
         in := in + 1
         contents[in mod size] := i
      end
end
```

```
class BUFFER-G3 inherit BUFFER-G
feature {NONE}
   afterPut : BOOLEAN
feature
   create is
      do
         afterPut := false
      end
   after-put : BOOLEAN is
      do
         Result := afterPut
      end
   get : BUFFER-ITEM is
      require
         not empty
      do
         afterPut := false
         Result := BUFFER-G::get
      end
   gget : BUFFER-ITEM is
      require
         not empty and not after-put
      do
         Result := get
      end
   put(i : BUFFER-ITEM) is
      require
         not full
      do
         afterPut := true
         BUFFER-G::put(I)
      end
end
```

Figure 30 – History-only Sensitivity Inheritance Anomaly.

The example they give is of a language with method guards in which it is desired to derive a class from the *BUFFER-G* class given above having a method called *gget*. This method is the same as *get* except that it may not be executed immediately after an execution of the *put* method. The sub-class defining the *gget* method, *BUFFER-G3*, is shown in Figure 30.

Inheritance is ineffective here as well: the *get* and *put* methods both need to be redefined in the sub-class. If accept sets were to be used instead of method guards, then it would be possible to inherit the *get* method implementation, only the *put* method would need to be redefined. Construction of a sub-class of *BUFFER-AS* to demonstrate the use of accept sets in this case is left as an exercise for the reader. In summary, both kinds of mechanism suffer from this form of the inheritance anomaly: method guards more so than do accept sets.

This being noted, it is by no means clear that this form of the inheritance anomaly is a significant problem in practice. Realistic examples resulting in this anomaly are not easily seen; certainly, the example given by Matsuoka *et. al.*, and repeated above, is hardly realistic.

```
class BUFFER-G
feature {NONE}
   in : INTEGER
   out : INTEGER
   size : INTEGER
   contents : ARRAY<ITEM>
feature
   create(s : INTEGER) is
      do
         size := s in := 0 out := 0
         contents :=
            new ARRAY<ITEM>(size)
      end
   empty : BOOLEAN is
      do
         Result := in = out
      end
   full : BOOLEAN is
      do
         Result := in = out+size
      end
   get : BUFFER-ITEM is
      require
         not empty
      do
         out := out + 1
         Result :=
            contents[out mod size]
      end
   put(i : BUFFER-ITEM) is
      require
         not full
      do
         in := in + 1
         contents[in mod size] := i
      end
end
```

```
class BUFFER-G4 inherit BUFFER-G
feature
   gget : BUFFER-ITEM is
      require
         not empty and after [^{put}]
      do
         Result := get
      end
end
```

Figure 31 – Solution of History-only Sensitivity Inheritance Anomaly.

If this form of anomaly is regarded as significant, then there is a simple extension of the method guard facility that will eliminate it. A single construct is added within method guards (pre-conditions). This has the form:

```
after method-trace-regexp-specification
```

This construct is a Boolean expression, which is true only if the suffix of the method trace of the object matches the given specification. These specifications may be given as regular expressions with method names as terminals, hence the pre-condition for the *gget* method in the example above is:

```
not empty and after [^{put}]
```

That is, when not empty and after any method other than *put*. Note the use of braces to delimit the terminal symbol (*put*).

Given this extension, the class *BUFFER-G4*, equivalent to the class *BUFFER-G3*, can be implemented without re-implementing the *put* and *get* methods. This is shown in Figure 31.

```
class BUFFER-G
feature {NONE}
   in : INTEGER
   out : INTEGER
   size : INTEGER
   contents : ARRAY<ITEM>
feature
   create(s : INTEGER) is
      do
         size := s in := 0 out := 0
         contents :=
            new ARRAY<ITEM>(size)
      end
   empty : BOOLEAN is
      do
         Result := in = out
      end
   full : BOOLEAN is
      do
         Result := in = out+size
      end
   get : BUFFER-ITEM is
      require
         not empty
      do
         out := out + 1
         Result :=
            contents[out mod size]
      end
   put(i : BUFFER-ITEM) is
      require
         not full
      do
         in := in + 1
         contents[in mod size] := i
      end
end
```

```
class LOCKED-BUFFER
   inherit LOCK,BUFFER-G
feature
   put(i : BUFFER-ITEM) is
      require
         not full and not locked
      do
         BUFFER-G::put(i)
      end
   get : BUFFER-ITEM is
      require
         not empty and not locked
      do
         Result := BUFFER-G::get
      end
end
```

Figure 32 – Inheritance Anomaly: The State Modification Case.

4.1.6.3.3 Inheritance Anomaly: Modification of Acceptable States

The third kind of inheritance anomaly is called 'modification of acceptable states' by Matsuoka *et. al.*. This occurs when two (or more) classes are combined using multiple inheritance, and the request scheduling semantics of methods of one of the base classes modify the request scheduling semantics of methods of another of the base classes.

An example of this is the combination of the *LOCK* class (shown in Figure 27 on page 96), with another class (the *BUFFER-G* class shown in Figure 29 on page 98 is used in this example). The semantics of the *LOCK* are that a call to the *lock* method prohibits any calls to any method until the *unlock* method is called. When *LOCK* is combined with a class such as *BUFFER-G*, it is desired to extend these semantics to prevent access to those methods inherited from *BUFFER-G* whilst the locking is in force.

The combined class, *LOCKED-BUFFER* is shown in Figure 32. Here it can be seen that it is necessary to redefine the implementation of the *get* and *put* methods inherited from the *BUFFER-G* class. This is an instance of the inheritance anomaly since the implementation of these methods is the same as it was for *BUFFER-G*. In addition to this, another problem is apparent. According to the rules given previously, a sub-type may not strengthen method pre-conditions, hence *LOCKED-BUFFER* is not a valid sub-type of *BUFFER-G* (though it is a sub-type of *LOCK*).

The first problem is easily solved by permitting the strengthening of an inherited method's pre-condition whilst retaining the inherited method implementation. Given that the inherited implementation was correct with respect to its original pre-condition, it will also be correct with respect to any strengthening of that pre-condition. The sub-typing problem remains, however.

The rules presented in the previous chapter governing the coordination of pre-conditions and sub-typing are based on the assumption that pre-conditions have failure semantics. Given such semantics, the pre-condition of any given method of a sub-type must be no stronger than the pre-condition of the corresponding method of the super-type. This is necessary to support the substitution of sub-type values where a super-type is expected. If this rule is not enforced, then the following situation may arise:

1. A variable, *v*, is used by a client programmer in a way that is fully compliant with the contract defined for the declared type, *T*, of that variable.
2. A value of a sub-type, *S*, of the type *T* is assigned to *v*.
3. The sub-type, *S*, strengthens one or more method pre-conditions, and consequently a contract violation (and a failure) may occur even in the case when the client is fully compliant with the information available to it.

Failures such as this represent a breakdown in the contractual basis adopted for object-oriented programming, and are therefore clearly unacceptable.

However, where pre-conditions have delay semantics, the situation may be different. There are two alternative views:

- A client using a sub-type object with strengthened pre-conditions via a reference of some super-type of that sub-type, may simply be subject to a longer than expected delay in having its request serviced. Consequently, the strengthening of those pre-conditions that have delay semantics may be allowed.
- The additional delay in the previous view may in fact introduce deadlock, consequently, even pre-conditions with delay semantics should not be able to be strengthened.

The first view is correct. The second view should be rejected because it is already possible for classes implementing a sub-type to redefine the behaviour of a given method in comparison to the behaviour of that method in classes implementing a super-type. This redefinition of behaviour may, in and of itself, introduce deadlocks. Consequently, the potential for strengthened pre-conditions to create deadlocks does not represent a new mode in which deadlocks may occur. Rather, it is simply an extension of the mode that already exists resulting from the ability to redefine method behaviour. In fact, deadlocks which occur as a result of pre-condition strengthening are likely to be easier to detect than are deadlocks resulting from method behaviour redefinition, simply because they are a restricted case.

Given that pre-condition strengthening is permitted when pre-conditions have delay semantics, language rules must be provided to ensure that clauses strengthening pre-conditions always have delay semantics. Here are two informal rules:

- Any clauses which strengthen pre-conditions must be clauses which have delay semantics given that they are used for a separate object, the rules determining that a given pre-condition has delay semantics having been given previously.
- To ensure that strengthened pre-conditions occur only for separate objects, any class containing a method having a strengthened pre-condition must be a ***separate*** class, for which all instances are ***separate*** objects.

Given these two rules, and the previous mechanism for strengthening pre-conditions without method redefinition, method guards can avoid this third kind of inheritance anomaly.

4.1.7 Reply Scheduling

The final area in which synchronisation issues must be addressed in a concurrent object-oriented programming language is termed reply scheduling. This is concerned with how the client handles the reply to a previously submitted request. This issue arises with the introduction of concurrency because (when the target of a method call is separate) it is possible for a client to submit a request for a method to be executed and then immediately continue, retrieving the result of this request at a later time. Reply scheduling is broken down into two parts:

- Retrieval of the reply value(s) returned by a previously submitted request.
- Synchronisation with the arrival (at the client) of the reply to a previously submitted request.

In each case, a reply-matching operation is undertaken, establishing the correspondence between the reply and a previously submitted request. Each of these can be viewed as providing the client with some information about a previously submitted method call. In the first case, the information is the result of that call. In the second case, it is ordering information (that the call is now complete).

After cursory inspection, it might be thought that this information is sufficient to support asynchronous calls to separate object methods. However, in the model of contract-based object-oriented programming advanced in the previous chapter, the return of a (local, synchronous) method call includes more information than just that given above. The complete list of information delivered is:

- Whether the call fulfilled its contract or not.
- In the case where the contract was fulfilled:
  - The call's result(s), if any.
  - The knowledge that the call has completed execution.
  - The knowledge that any calls made as part of the execution of the call have completed execution and fulfilled their contracts.
- In the case where the contract failed:
  - An indication of the type of failure (exception), including the name of the exception (and possibly an associated value).

A reply scheduling facility for object-oriented systems must provide the client with all of this information, not just some of it. In addition, it is important to realise that the information related to the success (or otherwise) of contract completion is delivered even when the method returns no data values.

4.1.7.1 Reply Scheduling in SCOOP and Eiffel //

The request scheduling mechanism in SCOOP follows that of Eiffel // very closely. The SCOOP mechanism is now examined, in order to illustrate the issues involved in request scheduling for object-oriented languages.

In SCOOP, reply scheduling for a separate method call has different semantics depending on whether or not the method returns a value. In the latter case, there is no reply, and consequently no reply scheduling. Separate calls to methods that do not return values are asynchronous from the client's point of view. For the case of methods returning values, the defined semantics upon issuing a separate call are to wait for the reply to that call to be returned. This is a completely synchronous semantics. However, the language implementor is permitted to relax this requirement for synchronisation along the lines pioneered by Caromel's wait-by-necessity mechanism [Caromel 1991, 1993].

```
put(x:G) is
  require
    not full
  ensure
    not empty
```

Figure 33 – An Example Method with a Postcondition.

The main idea behind wait-by-necessity is that synchronisation occurs at the (dynamically) first point where it is necessary to access the value(s) returned by a call, rather than at the point of that call. Consider the following:

```
† O1.x := so.get-count -- Separate call
‡ y := 67
§ z := O1.x.get-count + 45
```

Conventional (synchronous) reply scheduling would cause the client to wait at † for the reply to the call to *get-count*. However, the result of the call is not actually used in line †, all that happens is that the result (whatever it might be) is assigned to O1.x. Therefore, it is not necessary to wait at line †. Line ‡ is independent of the result, so neither is it necessary to wait at that line. At line §, there is a (local) method call on the object returned by the call, so it is here that the client must wait. In this way, wait-by-necessity allows continued execution of the client to be overlapped with the delay whilst the call is in progress. For completeness, wait-by-necessity applies to calls that do not return values, in that it is never necessary to wait for such calls. This is the same as the semantics defined for separate calls that do not return values in SCOOP. Hence, even a completely straightforward implementation of SCOOP makes use of wait-by-necessity.

One motivation for wait-by-necessity was to introduce extra asynchrony (and consequent enhanced performance) whilst not requiring significant modification of programs [Caromel 1993]. This should enable re-use of a large part of the code base of a sequential system in the implementation of an equivalent concurrent (or distributed) system.

4.1.7.2 Postconditions and Contract Fulfilment with Asynchrony

Recall the precondition paradox, which required precondition clauses involving separate objects to have wait semantics instead of the normal failure semantics. There is a somewhat similar situation is respect of the semantics of postconditions.

Concerning this issue, Meyer [Meyer 1997] writes as follows, in his discussion using the example method shown in Figure 33:

> *"There is a similar postcondition paradox: on return from a separate call to put we cannot any more be sure that **not** empty and other postcondition clauses hold for the client. These properties are satisfied just after the routine's termination; but some other client may invalidate them before the caller gets restarted. Because the problem is even more serious for preconditions, which determine the correct execution of suppliers, the rest of the discussion mainly concerns preconditions."*

Meyer does not offer any means to deal with this situation. This is probably appropriate: if the situation were as he describes it, in that when the client is restarted it is at least assured that the postcondition was true. In other words, that the call to *put* fulfilled its contract at some point in the client's past. The client's primary interest is in having the contract fulfilled, this interest is satisfied and the client can proceed with the knowledge that the service it requested has been provided. Any client interest in continuing truth of the postcondition, which may not be satisfied, is only at most only of secondary importance.

```
class QUEUE[ITEM]
  last : ITEM … -- returns last item on queue
  add(i : ITEM) is - adds item to queue
     do
        …
     ensure
        last = i -- i is at end of queue
     end
end
```

```
class TASK-QUEUE
  queue : QUEUE[TASK]
  count : INTEGER
  add-task(t : TASK) is
     do
        queue.add(t)
        count := count + 1
     end
  …
end
```

```
class PRINTER-INTERFACE
   tq : TASK-QUEUE
   print (pj : PRINT-JOB) is
      local
         t : TASK :=
            new PRINT-TASK(pj)
      do
         tq.add-task(t)
      end
end
…
† prntr-intf.print(p)
```

Figure 34 – Centralised Printer Queue.

However, Meyer is simply wrong in his assumption that the contract has been fulfilled. In fact, it is quite possible that when the client restarts, the contract associated with *put* has not been fulfilled because the execution of that method has not terminated, or may not even have begun. In other words, the contract's fulfilment may be in the client's future, or indeed, the contract may fail be fulfilled, due to circumstances beyond the method implementor's control, a network failure for example. This, of course, is a serious problem for the client, which may not be able to proceed until given an assurance that the remote contract has been fulfilled.

This issue is now explored with the aid of the example shown in Figure 34. When control returns from the call on the line marked †, it is known that the *print* method has fulfilled its contract (that is, that the print job has been put on the queue).

Now consider the modified example shown in Figure 35. The only substantive[61] difference is that the *TASK-QUEUE* has been declared separate. The *PRINTER-INTERFACE* class and the client use thereof are unchanged. When control returns to the client at ‡, the call to *add-task* at § may still be in progress. Since the contract for *print* includes successful completion of this call (*ie.* the post-condition of the call to *add-task* being true), it is not possible to say at the point marked ‡ that the contract for *print* has been fulfilled. Nor has the contract necessarily failed. Although the call (to *print*) has competed, the associated contract (instance) is in an indeterminate state. Furthermore, there is no way to tell when, if ever, the contract reaches either a fulfilled or failed state, nor can the exception raised in the case of a failure be determined.

The simplest way to return the contract instances in the above example to more useful semantics is to require synchronisation of all outstanding remote calls at the end of the (local) method in which they are made. The degree to which this reduces asynchrony is unclear.

[61] The introduction of the *real-add-task* method is necessary because of the SCOOP rule requiring any separate object used in a method to be a parameter of method.

```
class TASK-QUEUE
  queue : separate QUEUE[TASK]
  count : INTEGER
  real-add-task(q : separate QUEUE[TASK] t : TASK) is
     do
§        q.add(t)
     end
  add-task(t : TASK) is
     do
        real-add-task(queue,t)
        count := count + 1
     end
  ...
end
class PRINTER-INTERFACE -- Same as before
‡ prntr-intf.print(p)
```

Figure 35 – Distributed Printer Queue.

A second possibility is to rewrite the *real-add-task* method of the (second) *TASK-QUEUE* class as shown below:

```
real-add-task(q : separate TASK-QUEUE t : TASK) is
   do
      q.add(t)
   ensure
      q.last = t
   end
```

The semantics of calls involving separate references are such that the call to *q.acknowledge-addition* would cause the execution of *real-add-task* to wait for the call to return a value. At this point it would be known that *add* had fulfilled its contract because the semantics require that the object attached to *q* execute the calls to *add* and *last* in the order they are sent. If the execution of *add* terminated abnormally, the exception thus triggered would be propagated to the caller (*real-add-task*) in the normal fashion. Typical environments for languages with assertion support (such as Eiffel) allow the checking of post-conditions to be suppressed once a module has passed the testing phase. To support the idiom used above, the system would have to avoid suppressing evaluation of post-conditions that involve separate references. This is analogous to the case for pre-conditions involving separate references. This approach can be termed the *separate post-condition technique*.

This technique has the problem that only the caller of remote method can synchronise with its completion. There are other alternatives, which allow programmers to obtain information about asynchronous contracts whilst they may still be in progress. This issue is discussed further, later in the thesis.

4.2 Distribution Specific Issues

This section discusses how the core object model described above should be mapped to the distributed environment to construct a distributed object system.

4.2.1 Proxies

In a distributed system, the methods of a remote object cannot be called directly. Therefore, most distributed object systems use special objects, typically termed *proxies*, as local representatives for remote objects. The role of these proxies is simply to support indirect access to the methods of the remote object they represent. For example, a method call directed to a proxy is forwarded via an RPC-like mechanism to that proxy's master object.

4.2.2 Failure Semantics of Method Calls to Remote Objects

As discussed in Chapter Two, the fact that processes in distributed systems fail independently has implications for the reliability of communication. This thesis has adopted at-most-once semantics for message transport. The implication of this is that method calls on remote objects behave differently to method calls on local objects.

A language mechanism should have as a goal the restriction of the differences to the minimum necessary. Furthermore, the language should assist in restricting the scope of changes to source code necessary to transform a centralised system to a functionally equivalent distributed system.

The approach taken in Java is to require the signature of any remote callable method to include an indication that a system defined exception may be raised in the course of the method's execution. Raising of this exception indicates possible failure (of the network or server process) whilst the call was in progress.

In Java, the code modifications required to convert a call to a local object into a call to a remote object can be often confined to the addition of an exception handler dealing with cases where a possible failure is indicated. These cases are genuine exceptions, in that they occur rarely and are due to events outside programmer control, so this is not an abuse of the exception handling mechanism. In the majority of remote calls, where the exception is not raised, the semantics of remote calls with respect to call failure are the same as for local calls, in that the call executes exactly once.

Apart from the semantics of failures, there is a second way in which remote calls commonly differ from local calls, namely the mode of parameter transmission. This is discussed in Section 4.2.4.

4.2.3 Run-time Binding

One characteristic of a distributed system distinguishing it from both sequential and centralised concurrent systems is that different parts of the system may be created independently. There is no single root element responsible (directly or indirectly) for the creation of all others. Given this, it may often be necessary for a part of the system to establish contact with another part of the system at run-time.

Establishing contact with another part of the system is termed *run-time binding*[62]. Like all bindings, such as the binding between a formal parameter and the associated actual parameter to a method call, these run time bindings must be strongly typed. This requires some degree of dynamic type checking.

Run-time binding in the Napier88 persistent programming language [Morrison *et.al* 1989] is provided by a special language type called the environment (denoted **env**). In Napier88, bindings from environments are available within a special kind of program block, the **use**...***with***...**in** block, as shown in Figure 36. The type checking of the use of *a* and *b* within the block is static. Dynamic checks that the bindings named *a* and *b* in the environment named *an_env* are in fact of the correct type occur at block entry and the program exits if any of the checks fail. This is compatible with the principle requiring clustering of dynamic checks from Section 2.3.1.2. Additional facilities are provided to create environments, to insert bindings into them and to remove bindings from them.

The environment construct was actually designed for use in another, non-distributed, context: establishing bindings to objects in a persistent store. Discussion of persistence, whilst both important and interesting, is outside the scope of this thesis.

One flaw in the Napier88 mechanism stems from the language's blind pursuit of orthogonality in design, which leads to **use**...**with**...**in** block being permitted within arbitrary language constructs

[62] The term *run-time binding* is adopted to distinguish this facility from dynamic binding, which, in being the binding of an object's methods to that object at the point of its creation, is also a run-time occurrence.

```
use an-env with a : A-TYPE; b : ANOTHER-TYPE in
  a.m1(b) -- Use of run-time bound values.
  ...        -- Additional code using values.
```

Figure 36 – Napier88 Environments.

```
constructor65([parameters]) is
  [require preconditons]
  [bind
    {from namespace use {name [as localname] : type}+}+]
  (syntax for reminder of constructor method definition)
```

Figure 37 – Run-time Binding Syntax.

(even expressions). In allowing this, the language provides less support for the clustering principle than would otherwise be the case. Inexperienced Napier88 programmers have been observed to produce programs wherein run-time type checks take place for every activation of a procedure, whereas the checks could equally validly be performed a single time when the procedure is defined[63].

To avoid this problem, it is proposed that dynamic binding be permitted only within an optional clause in the definition of constructor methods, denoted by the **bind** reserved word. The syntax is shown in Figure 37. This syntax supports binding to one or more objects in one or more namespaces. The binding attempts are checked in the order they are written, and an exception is raised to the caller of the constructor if any of the attempts fails. Names in the **use** list introduce new local variable bindings. Namespaces are objects, so the namespace used in a later **from** sub-clause be a binding established in an earlier **from** sub-clause in the same constructor. In addition to being used in this clause, namespaces would support operations to add and remove bindings dynamically. The syntax ensures that dynamic binding complete prior to any side-effect producing computation in the constructor[64].

This construct ensures that, for any given object[66] (module instance), any dynamic binding required by that object is clustered at the beginning of that object's lifetime. Once an object is successfully constructed, the execution of any method on that object is statically safe.

4.2.3.1 Imprecise Binding

Dynamic binding can be imprecise, that is, an object can bind a reference to a remote object such that the type of the reference is a super-type of the type of the remote object. Consider a remote object with actual type *S*. Figure 38 shows two approaches to obtaining a binding to this object as some super-type, *T*, of *S*. In the first case, the client obtains a binding of type *S* and then immediately assigns to a variable of type *T* using a polymorphic assignment operation. The second case is crucially different, here the client simply requests a binding of type *S* directly.

The crucial difference is that in the first case the sub-typing abstraction is performed in the binder, whereas in the second case the abstraction may be done at the target of the binding. In the first case must the language used to implement the client be able to represent both types (*S* and *T*). However,

---

63 In the Napier88 persistent programming system, the interpretation of "the procedure is defined" is "the procedure is placed into the persistent store".

64 The evaluation of pre-conditions may occur prior to the dynamic binding, but pre-conditions are required to be side-effect free.

65 The ***constructor*** reserved word is not Eiffel syntax. In Eiffel, constructors are designated in a separate syntactic region of the class (the ***creation*** region) and otherwise follow the syntax of ordinary methods.

66 In a language with class-wide constructors, there would be a similar clause establishing bindings for class-wide use.

```
constructor use-remote-object(nameserver : NAMESPACE) is
  bind from nameserver
     -- Precise binding.to actual type of object.
     use remote-object as act-object : S
  local
     -- Polymorphic assignment.
     abs-obj : separate T := act-obj
  do
     ... Use abs-obj in some way ...
  end

constructor use-remote-object() is
  bind from nameserver
     -- Imprecise binding to super-type of object.
     use remote-object as abs-object : T
  do
     ... Use abs-obj in some way ...
  end
```

Figure 38 – Precise and Imprecise Binding.

the second case the client need not be able to represent the type *S*[67], this type need only be understood by the namespace. This distinction is important in linguistically heterogeneous systems, where there may be several differing type systems (with some common ground).

4.2.4 Objects as Parameters to Remote Method Invocations

In passing parameters to remote method invocations, there are really only two choices in respect of language semantics:

- *Call-by-value* parameter transmission.
- *Call-by-reference* parameter transmission.

Some apparent alternatives from the literature, such as *call-by-move* [Black *et. al.* 1986, Achauer 1993], are really an implementation technique for one of the above. Another set of alternatives from centralised programming languages, such as *call-by-name* (macro-substitution), cannot be implemented in the distributed context.

For parameters of basic types, such as integers and floating point numbers, call-by value should be used, this follows standard practice in centralised programming languages and is easy to implement in the distributed context. For objects however, there is a real choice to be made (or it may be possible to support both options). The two possibilities are explored below.

4.2.4.1 Passing Objects by Reference

If a reference to a object is passed over a network, then that reference must be meaningful in both the source and destination processes. In practice, because the destination might then pass the object off to a third process, object references should be made meaningful in all processes. Typically, object references include either a network address, or some name that is mapped to a network address by a system service.

In most object systems, object references are object proxies, so operations applied to an object reference are applied, via a network protocol, to the master object that is referred to. Given a proxy mechanism, support for passing objects by reference is a simple addition (proxies are passed, by value, between processes in some system defined way that is not necessarily applicable to ordinary

[67] Note however, that the client must trust the namespace to enforce the sub-typing relation correctly. This trust relationship carries security implications.

objects). The drawback, of course, is that any operations on the object via its proxies are delayed by the network latency.

4.2.4.2 Passing Objects by Value

Due to the latency of operating on an object passed by reference, it is also desirable to be able to pass objects by value, thereby enabling method calls issued at the destination to avoid the network latency.

It is problematic to support passing objects by value in a linguistically heterogeneous object system. The problem is that the state of an object includes[68] the code of that objects methods. Transmission of an object by value must therefore ensure that the code of that object's methods is available at the receiving end. If the sending and receiving ends are implemented in different programming languages, the version of that code used in the sender can not be used in the receiver. There are three possible alternatives for a solution:

1. Automatically convert the code of the method implementation in the sender into equivalent code in the receiver.
2. Substitute method implementations written in the programming language of the receiver when the object is received.
3. Prohibit transmission by value of any class of objects defining methods, with the exception of constructors, since these will not be executed by the receiver as the object it receives is already constructed.

In the current versions of CORBA [OMG 1997], the issue is avoided by not providing support for the passing of objects by value. Recent work has led to a number of proposed object-by-value extensions, in general, these follow alternative 2.

The first alternative is infeasible, and is identified as such in Section 2.4.1.3. To support this alternative would require a bidirectional mapping of each programming language (in the system) to and from some common language, or alternative bidirectional mapping between each pair of languages, including each language's run-time environment. Even if this could be done in principle (which is unlikely) it would be prohibitively expensive in practice.

The third alternative is straightforward but unreasonably limiting. In particular, it prevents the application of polymorphic method calls to objects passed by value.

The second alternative is adopted in this thesis. When an object is received, the system locates and substitutes implementations of that object's methods as part of the unmarshalling process. An exceptions is raised if an appropriate implementation cannot be found. The are two ways in which object may be transmitted:

- In the parameters passed to a remote method invocation.
- In the transitive closure of the result of a remote method invocation.

Both of these cases can lead to the raising of an exception indicating failure to locate an appropriate method implementation:

- Reception of an object in the transitive closure of the parameters for a call without an appropriate implementation will be detected when the server for that call attempts to unmarshal the request containing those parameters. In this case, an exception is propagated back to the caller and the method is not executed.

68 The state of an object includes one or more references to the methods defined in the class of that object. These references are necessary to support polymorphic method calls. In typical systems, these references are pointers to arrays of functions, although some higher level tag might also be used.

- Reception of an object in the transitive closure of the result of a call without an appropriate implementation will be detected when the caller attempts to unmarshal the reply containing that result. In this case also, an exception is raised in the caller but the method has been executed.

In the second case, the remote method must be a query since it returns a result. The restrictions given in Section 3.3.1.2 prohibit queries from producing side-effects on abstract state. Therefore, the fact that the call has executed has had no effect on the abstract state of the server. Consequently, in neither case where the call fails due to lack of a class implementation is there any need for corrective action to be taken on the server.

4.2.4.2.1 Unmarshalling and Invariants

In sequential object systems with linguistic support for assertions, class invariants are verified at object creation, and at every method call on each object. Consider a case where an object is passed by value from one node to another, where the latter node is implemented in language supporting assertions but the former is not. When the object is received, there is no guarantee that its state is subject to the invariant. Consequently, the unmarshalling on the latter node must be check class invariants on all objects. This leads to a second kind of exception that may be raised during unmarshalling, this indicates invariant violation and is raised in the same way as the exception indicating lack of an appropriate implementation.

4.2.4.2.2 Security Implications

There are security implications of the substitution of class implementations and the transmission of objects by value:

- The state of objects received by value cannot be trusted to the same extent as that of object created locally. This is the case even if the object received by value conforms to the appropriate invariants, because these invariants express general correctness properties of the object rather than incorporating security restrictions.
- There may be a mechanism to load class implementations over the network, as there is in Java [Arnold and Gosling 1997]. This obviously has security implications, and it must be possible to restrict or prevent such class loading and to constrain the execution of code thus loaded (by containing that execution in a sandbox).

4.2.4.3 Choice of Object Transmission Mode

Given that both modes for object transmission can be supported, there is a need for programmers to control which mode is used in a given instance.

In Eiffel //, objects of classes descended from class *PROCESS* are passed by reference and all other objects are passed by value. In SCCOP, all objects are passed by reference and are attached to a **separate** reference at their destination. However, there is a special mechanism to import remote object closures attached to separate references, supporting transmission of object graphs by value.

The SCOOP approach is preferable in principle, in that the parameter transmission mode may be determined on a per-call basis, rather than being determined on a class-wide basis. However, this approach requires language support (the **separate** reserved word as an attribute of references, and its associated semantics) which is not a common object-oriented mechanism. Therefore, the SCOOP approach is unsuitable for middleware in linguistically heterogenous systems.

The approach adopted in the Java RMI system is similar to that of Eiffel //, in that objects implementing a particular interface (*Remote*) are transmitted by reference and all other objects transmitted by value. However, there are some differences between the Eiffel // *PROCESS* class and the Java *Remote* interface:

- Unlike *PROCESS* objects, classes implementing *Remote* do not incorporate an intrinsic thread of control.

- Unlike *PROCESS* objects, classes implementing *Remote* do not partition the objects of other classes in the system. In particular, a given non-Remote object may be reachable in the local[69] transitive closure of two or more Remote objects.

In effect, the *PROCESS* class specifies three properties: remoteness, execution and partitioning, whereas the *Remote* interface only specifies remoteness. In this, it is similar to separate references without requiring compiler support. As such, a class (interface) such as *Remote* is more appropriate for controlling parameter transmission in heterogeneous object systems. One might, of course, augment the *Remote* interface with a *Process* interface derived from it and having the additional properties of the Eiffel // *PROCESS* class. However, these properties do not concern communication and consequently are outside the scope of this thesis.

4.2.5 Distributed Object Services

The Object Management Group (OMG) has defined an extensive collection of common object services [OMG 1998] on top of CORBA. These include the following:

- *Naming Service* – supports the binding of names to objects. These names are relative to particular naming contexts. Each naming context is a member of a graph of naming contexts which may be distributed across the system.
- *Event Service* – supports the routing of application defined events between one or more producers and zero or more consumers. Producers and consumers need not be known to one another. Both push and pull models of interaction between the event service and both producers and consumers are supported.
- *Life Cycle Service* – supports creation, deletion, migration and cloning of objects.
- *Persistent Object Service* – supports persistent storage of objects.
- *Transaction Service* – supports transactions as a means of concurrency control over operations on shared objects. Transactions may be implicit or explicitly managed by the methods of objects. Optionally, an implementation of this service may support nested transactions.
- *Concurrency Control Service* – supports low level concurrency controls to allow cooperating concurrent threads to coordinate access to shared objects.
- *Relationship Service* – supports the entity-relationship model for relationships between objects (entities). Relationships can be of arbitrary degree and can be constrained as to the type and cardinality of member objects (entities). Relationships can be further elaborated by the assignment of specific roles to the member objects.
- *Externalization Service* – supports the serialisation of objects into (byte) streams, and subsequent deserialisation of streams containing this serialised form into objects.
- *Query Service* – supports queries to obtain information about collections of objects.
- *Licensing Service* – supports licensing of access to objects.
- *Property Service* – supports the dynamic association of named properties with objects. Includes mechanisms for operating on sets of properties in a single operation.
- *Time Service* – provides clients with access to timing information, including the ability to establish partial ordering on sets of events.
- *Security Service* – supports secure interaction between objects. Provides for identification and authentication of principals. Supports authorisation and access control over operations. Provides auditing of security related operations. Supports secure communication between

[69] The local transitive closure is the transitive closure exclusive of references to remote objects.

objects. Ensures non-repudiation of actions undertaken by (or on behalf of) principals. Finally, provides for management of the security environment.

- *Object Trader Service* – provides a matchmaking service between objects! Specifically, allows clients needing a particular service to obtain information about servers offering to provide that service (or a potentially substitutable service).
- *Object Collections Service* – supports a standard set of abstractions over groups of objects. example abstractions include stacks, queues, lists, sets and trees.

This thesis will follow the OMG approach of assuming that most object services can be described in terms of the middleware, rather than requiring extension to that middleware, and will consequently avoid further discussion of object services issues.

Chapter Five

Framework for Analysis

This chapter consolidates the theory in the previous three chapters to develop a practical framework within which to analyse the effectiveness of middleware in distributed systems. The framework is also useful as a tool in the design of new middleware. The framework consists of three parts:

- ***Application*** – the effectiveness of middleware is limited by the range of tasks to which it can be applied.
- ***Software Engineering*** – support for modern software engineering practice has been the primary factor driving general purpose programming language design in the past two decades. In order to be useful to the modern software development process, a middleware system must provide support for software engineering.
- ***Performance*** – a middleware system can only be considered effective if it provides performance comparable, or superior, to current communication mechanisms.

5.1 Analysis of Application Issues

The application component of the framework is developed in five stages:

1. Identification of a core model describing useful interactions in distributed systems (Section 5.1.1).
2. Identification of a core model describing the role of high level middleware in distributed systems (Section 5.1.2).
3. Enumeration of the kinds of processes in distributed object based systems (or distributed systems in general) and characterisation of the kinds of interaction that occur between them (Section 5.1.3).
4. Determination of the support that the core interaction model and each specific sort of interaction requires from the middleware, leading to the development of a checklist which may be used in the assessment of this support (Section 5.1.4).
5. Development of a set of constraints on the mechanisms admissible for assessment within the overall framework is applicable, defined in terms of application issues (Section 5.1.4.8.2).

5.1.1 Core Interaction Model

Different communication mechanisms may exhibit different structures in message exchange. For example, Remote Procedure Call (RPC) [Nelson 1981, Birell and Nelson 1984], exhibits a message structure in which every request message has a corresponding reply, whereas some other communication mechanisms do not entail this one-to-one correspondence. In order to compare and assess different communication mechanisms it is necessary to abstract away from concepts that are specific to a particular class of communication mechanism. This leads to a more general notion of interaction in distributed systems. In order to be useful in an environment in which failures occur, this generalised concept of interaction needs to exhibit certain properties, which are also identified.

5.1.1.1 Generalised Interactions

In this framework, an interaction consists of the execution of a number of *events* by a number of processes in a distributed system and *information transfers* between processes as needed to coordinate the execution of these events. An event cannot occur until all information transfers into that event have occurred; information transfers out of an event occur only after the execution of that event. An interaction has a single event of origin: all other events are caused by this initial event. The initial event may inject information into the interaction from an outside source. This information corresponds to parameter values passed in from computations external to the interaction. After the successful completion of an interaction, the processes involved will each have obtained zero or more interaction results, of the following kinds:

- Side effects on local state.
- Result values, which may be used by the process receiving it in computations external to the interaction.
- The knowledge that all other results of the interaction have been successfully effected. This result must be provided to one process in the interaction. This process is called the termination coordinator for the interaction.

The termination coordinator is necessary because failures can occur, and consequently the interaction may not complete. It is the responsibility of the termination coordinator to detect definite success or possible failure[70] of the interaction as a whole and to take corrective action in the event that possible failure is detected.

Events are more than just computations; because of causality constraints, the need to model result values and the need to support coordinated termination; additional kinds of events are required. There are four kinds of events:

- ***The Initial Event*** – this (singleton) event has two purposes, it acts as the cause of all other events and it acts as the point at which parameter information may be injected into the interaction.
- ***Computation Events*** – these events correspond to the execution of code in the processes involved in the interaction. This execution acts on inputs of two kinds:
  - Inputs delivered by the information transfer that triggered the execution of the event.
  - Resources and state local to the process executing the event.

  The event may produce side effects on the local state and resources, and at the end of its execution, produces one or more information transfers to trigger other events. To simplify modelling, the framework allows a third kind of pseudo-input:
  - Constant information global to all processes in the interaction, in particular, executable code (subject to heterogeneity considerations)[71].
- ***Final Events*** – one of these events is executed on all processes except the termination coordinator. It must be the last event executed on a given process. The final event on a process acts as a sink for any result values that the interaction delivers to that process.
- ***The Terminal Event*** – this singleton event is executed on the termination coordinator and acts as the sink for any result values delivered to that process. In addition, it has the property that it

[70] Typically, an implementation detects the possible failure of an interaction using timeouts (as failure detectors).

[71] This kind of input is actually covered by the first kind of input. In principle all the constant information needed by any processes in the interaction can be injected as a parameter to the initial event and from there passed to all other events in information transfers. In practice, this convolutes the use of the model and so this alternative is provided. Note also that at the implementation level this information is likely to be made available in some special way: for example via a specialised, global address space; rather than being sent in all information transfers.

executes after *all other events* in the interaction, not just those that execute on the termination coordinator. Hence, execution of the terminal event corresponds to successful completion of the interaction.

An interaction can be described by a directed *interaction graph* in which events are vertices and information transfers are edges. Edges that connect events executing on different processes are implemented by message sends. There are a number of constraints on the topology of interaction graphs:

- The graph must be acyclic. This is necessary to avoid permitting an event to be its own cause, which is obviously nonsense.
- All events in the graph must be in the transitive closure of the initial event. This is necessary because this event is the cause of all the others.
- The final event for a process must be in the transitive closure of all events executed on that process.
- The terminal event must be in the closure of all events executing on the termination coordinator.
- One of the events on the termination coordinator is referred to as the *prepare-for-termination* event. All final events in the interaction must be in the transitive closure of this event.
- There must be an edge from each final event to the terminal event.

The last four constraints are required to provide coordinated termination.

This model describes only interactions that complete successfully. To be more precise, the model describes interactions that are *assured* to have completed successfully. As described in the introduction, this thesis is not concerned with recovery from failures but is interested in ensuring that detection of failures (and consequently recovery) can be integrated with the mechanism for interaction.

The way to integrate failures handling is to place a limit on the amount of time that the termination coordinator will wait for the interaction to terminate successfully. The interaction is then declared failed if the time limit expires, whereupon the termination coordinator may then take corrective action. This is implemented by starting a timer at the prepare-for-termination event (this is a failure detector, as described in Chapter 2). Hence, it can be seen that coordinated termination provides the essential infrastructure on which to build failure management at the interaction level. Additional timers may be used to manage the failure (and restart) of parts of an interaction. This additional complexity optimises failure recovery, but it is not essential and therefore is omitted from the model.

5.1.1.2 Interactions: Examples and Counter-examples

Consider the example of a single RPC involving a client and a server, as shown in Figure 39. Zero or more side effects are produced on the server and zero or more result values are returned from server to client. The client is the termination coordinator: the reply it receives from the server indicates that the server's involvement in the interaction has successfully completed. In addition, this reply typically contains the result values, if any, sent from server to client. The server is the only process other than the termination coordinator in the interaction. Two messages are required, these occur where information transfers cross process boundaries. Hence, it is clearly demonstrated that a single RPC satisfies the criteria for an interaction.

Now consider a slightly modified example (Figure 40), where the execution of the remote operation on the server, *A*, involves an RPC to some other server, *B*. In this case there are three processes involved in the interaction. The client (which is still the termination coordinator), must receive termination information from both of them. However, at the time when the first server sends its reply to the client it (the server) has already received termination information from the other server.

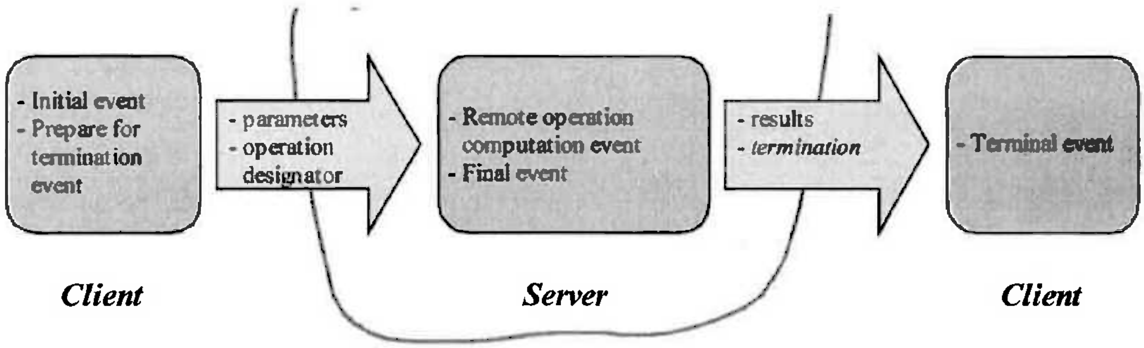

Figure 39 – Interaction Graph for a Single RPC.

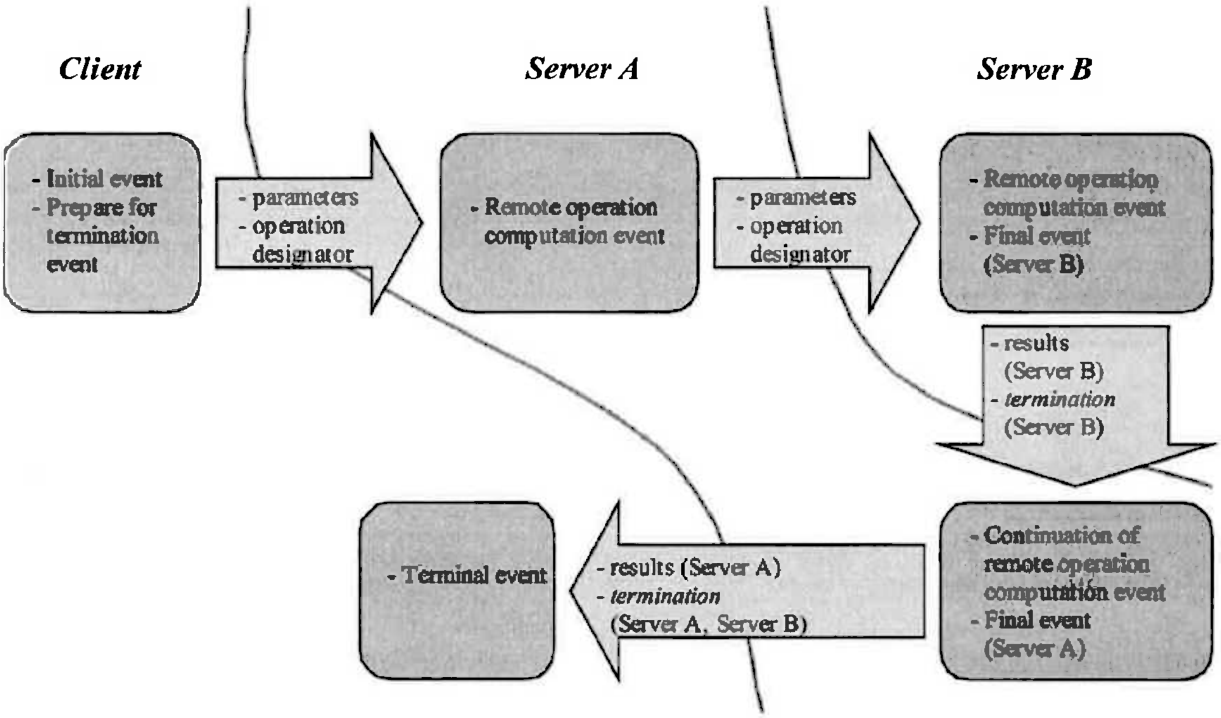

Figure 40 – Interaction Graph for Nested RPCs.

Consequently, the reply from the first server to the client contains a termination indication for both of the servers: no additional messages are required to establish interaction termination.

This example also demonstrates the possibility of localised failure management. If server A is unable to obtain a reply from server *B*, it can try another server rather than simply sending a message back to the client indicating a failure. In this way, it is sometimes possible to avoid having to restart the interaction from the beginning. The ability to do localised failure management is not a fundamental property of interactions: local recovery should be equivalent to restarting the interaction from the beginning except that it is usually faster.

Some RPC-based communications mechanisms allow for so-called "one-way" or "maybe" RPCs. These consist of a request message and a remote operation executed in response to this message, but no reply or acknowledgment message. Taken in isolation; these are not interactions: the reason is that neither client nor server can be the termination coordinator:

- The client cannot be the termination coordinator since it does not receive any information at all from the server and therefore can not receive a termination indication from the server.

- The server cannot be the termination coordinator either. Although it can receive information from the server, it cannot execute the prepare-for-termination event prior to the client executing its final event.

One-way operations are potentially still useful as parts of larger interactions. For example, a client/server interaction may consist of a one-way RPC followed by a normal RPC. The reply of the (second) RPC indicates termination of both RPCs when it arrives at the client, which acts as the termination coordinator.

5.1.2 The Role of Middleware

The role of middleware is to act as a bridge providing communication between computational elements of distributed systems[72]. In linguistically heterogeneous systems, such as those discussed in this thesis, these computational elements may be implemented in several different programming languages. An important component of the middleware role is the specification of element interfaces in some language that is neutral with respect to the element implementation languages. This language is typically termed an *interface definition language* (IDL).

Middleware remote operation mechanisms provide two interfaces, a client side interface and a server side interface. The client-side interface accepts requests from clients and delivers results to those clients. Typically, clients interact with this interface by calling methods on objects exported by the middleware. The server-side interface delivers requests to servers and accepts results from those requests for transport back to the client originating the request. Typically, the interaction across this interface occurs via calls made by the middleware to methods of server objects, these calls are generally termed *up-calls*.

To support high-level programming of communication, middleware communication facilities should be mapped to equivalent implementation language constructs in a seamless way. For example, client side facilities for remote method invocation should be mapped to (local) method calls on objects provided by the middleware. These local method calls should be expressed using the standard method call constructs for the client implementation language. For example, in a client implemented in Ada95, one would write:

```
m1(s,p1,p2)
```

In contrast, the equivalent call in Java, C++ or Eiffel is:

```
s.m1(p1,p2)
```

Middleware systems typically provide a compiler to translate IDL specifications into client side objects and methods in each client implementation language, thereby supporting the seamless interface shown above. Similarly, on the server side, IDL specification may be used to create wrapper objects to support seamlessness in the middleware/server interface. The fundamental point is that the middleware provides an interface that is adapted to support each implementation language, and developers continue to use existing implementation language constructs and compilers.

This framework is designed to assess communication mechanisms in the context of middleware systems. This does not preclude its application to communication mechanisms embedded within a programming language environment[73], provided such communication mechanisms can feasibly be extracted from the particular language of origin and made available in middleware form.

72 Middleware systems typically support other facilities in addition to communication, for example a facility for binding to remote servers in order to communicate with them. This framework ignores these other facilities, concentrating only on communication operations.

73 It is fortunate that such apparently language specific mechanisms can be included in the assessment, since many of the interesting mechanisms, particularly in the research community, existing only within single language environments.

5.1.3 Kinds of Processes

Andrews [Andrews 1991], suggests that processes in message-passing systems are of four kinds, as follows:

- *Clients*
- *Servers*
- *Filters*
- *Peers*

Each of these four categories is described below, and further categories introduced as necessary.

5.1.3.1 Clients and Servers

Clients and servers are considered together. These are by far the most common types of processes; being the key components of many recent computing systems structured as client/server systems.

Users generally interact with a distributed system via clients. For example, clients in a database system might parse queries typed in by a user and perform remote communications with other processes (typically servers) in order to effect the queries. The key distinguishing characteristic of a client is it is the initiator of remote communication, rather than responding to communication initiated by other objects.

Servers are objects that exist solely to provide services for other objects. Hence, pure servers do not initiate communication, instead they respond to communication initiated by other objects (typically clients). Servers in a client-server database management system are responsible for data storage, concurrency control and query evaluation. A slightly non-obvious example of a server is the process providing X window system functionality on an appropriate hardware device.

5.1.3.1.1 Client/Server Communication

Communication between clients and servers is characterised by request/reply structure. Clients send requests to servers, these then execute the requested remote operation (with parameters specified in the request) and send a reply containing zero or more results back to the client.

As discussed in the previous chapter, there is a simple mapping of client/server communication to object-oriented systems. An object (class) might declare a function returning the sum of two integers as follows:

```
add_integers(lhs,rhs:INTEGER): INTEGER
```

A client having a reference to an object, *s1*, of the class declaring this function would invoke it as follows:

```
sum := s1.add_integers(3,78)
```

That is, using exactly the same syntax as for a non-distributed object system.

It is often the case that in the history of interaction between a client and a server (process), the parameters to some requests are results of previous requests by that client to that server. In fact it may be the case that the server exports operations for which the only valid parameters are results returned by other operations exported by that server. Recall that in object-oriented programming, the target (object) of an operation is implicitly both a parameter, passed to that operation, and a result, returned by that operation.

One example of related requests is the navigation of a data structure held at the server. Consider the following object types[74]:

[74] This example is adapted from that given in [Bogle and Liskov 1994].

```
class DATABASE
  get-list(name : STRING): LIST
end

class LIST
  next              -- Advances to the next element in the list.
  value: INTEGER    -- Value of current element in list.
end
```

A client could retrieve the tenth integer in the list by executing the following:

```
l := db.get-list("myList")
for I in 2...10 loop
  l.next
end
the-tenth-int := l.value
```

5.1.3.1.2 Layered Servers

An immediate extension of the simple client/server classification is an process which is both the server for a group of processes and the client of another (distinct) group of processes. Such a process is termed a *layered server*. As discussed in Section 2.3.1.4, support for layered systems, in particular for three-tier layered systems, is an important software engineering and application requirement.

Distributed services are often organised as a collection of layered services. For example, a database management system might consist of servers in three layers:

- Top layer servers provide full DBMS functionality, including concurrency control.
- Middle layer servers provide stable storage of data and logs.
- Bottom layer servers provide access to data storage hardware devices.

Each layer apart from the last uses services of the layer immediately below in providing the services required of it. Clients interact directly only with the top layer. This arrangement has obvious advantages in terms of modularity, and, in particular, it is possible to add servers to the layers whilst the system continues to run. Returning to the database example:

- one might add a new bottom layer process in order to support a newly purchased RAID, or
- one might add top layer servers to provide concurrency control customised for the requirements of a particular client application.

A structure consisting of layered servers is a foundational concept for programming with systems such as Argus [Liskov 1981].

There are two important characteristics in the patterns of communication occurring in a system structured as a collection of layered servers:

- The remote invocation graph is acyclic – servers in a given level may invoke lower level servers, but do not invoke higher level servers or call other servers in the same level.
- Whilst servicing a remote operation, a server may receive further invocations. In, particular, the server may receive an invocation from a client or higher level server whilst waiting for completion of an invocation that it initiated on a lower level server.

5.1.3.1.3 Administrators

Gentleman [Gentleman 1981], introduces the concept of an *administrator*, a process providing a server like interface to clients but which does not itself implement any services. Instead, an administrator manages a pool of servers, client requests are directed to an appropriate server in the pool and the required service is executed by the selected server.

The patterns of communication in a client/server system incorporating one or more administrators are generally similar to those in systems without administrators. From the point of view of a client, an administrator resembles a server, and, from the point of view of the server, the administrator behaves like a client. In effect, administrators are simply "glue" permitting the interaction of clients and servers whilst hiding the servers behind a layer of abstraction.

It is worth noting that, whereas the client's request must reach the server via the administrator, the server's reply need not generally travel via the administrator. Instead, it may travel directly back to the client. This reply "short-cutting" effect can become particularly marked in a system involving several levels of administrators, for example a hierarchical name service such as DNS [Mockapetris 1987] could be implemented this manner. Hence, the patterns of communication observed in a system with administrators are similar to those in a layered server system, with the addition of reply "short-cutting".

Administrators are known under several additional names, sometimes they are called brokers, sometimes traders, and sometimes routers. This thesis uses administrator in preference to these terms, which have additional semantics appertaining.

5.1.3.2 Filters

Filters receive a stream of inputs via remote communications from one or more source objects. The filter transforms the input stream in some way. Finally, the transformed stream is output to one or more sink objects via remote communication. The three sub-systems within a filter generally run in a parallel or coroutine-like fashion. Filters are often used in text processing, for example, a filter that replaces each TAB character by a certain number of SPACE characters. The interior members[75] of a Unix pipeline are usually filters. Unix pipelines are an effective demonstration of the power of filters connected together in pipelines, thereby providing composition of the transformations effected by the filter elements.

5.1.3.2.1 Filter Communication

In many simple filter-based systems, such as in Unix pipelines, the only communication is between adjacent filters in the source-to-sink (or downstream) direction. In production-quality distributed systems this is rarely satisfactory. Usually it is also necessary to provide for communication in the upstream direction between adjacent filters, in order to signal errors in the received data stream and possibly also for flow control.

The data sent downstream is usually quantised in some way, for example, in fixed size records in a stream of binary data, or as lines separated by carriage returns in a stream of text. These discrete units are termed *packets*. The process of passing a single packet from a source, through a single filter and to a sink can be considered an interaction as shown in Figure 41. Note that, in order to provide termination coordination, there is feedback from the sink to the source (in the upstream direction). In the more general case, there may be more filters in the pipeline, and more than one packet may be processed. In this case, the sink only has to provide feedback to the source after it has processed the last packet[76].

There is often a need for additional upstream feedback in a filter pipeline. If a data packet passed downstream is invalid in some way, then the filter detecting the invalidity needs some way to signal this invalidity. The most usual approach is for the filter to send an error message (or exception) to the immediately upstream pipeline element. If the error cannot be dealt with by the element receiving it, it

[75] The interior members include all but the first (source) and last (sink) members of the pipeline.

[76] A possible implementation strategy is for the source to tag packets that require this acknowledgment feedback. This tagging would be passed through unaltered by all the filters in the pipeline.

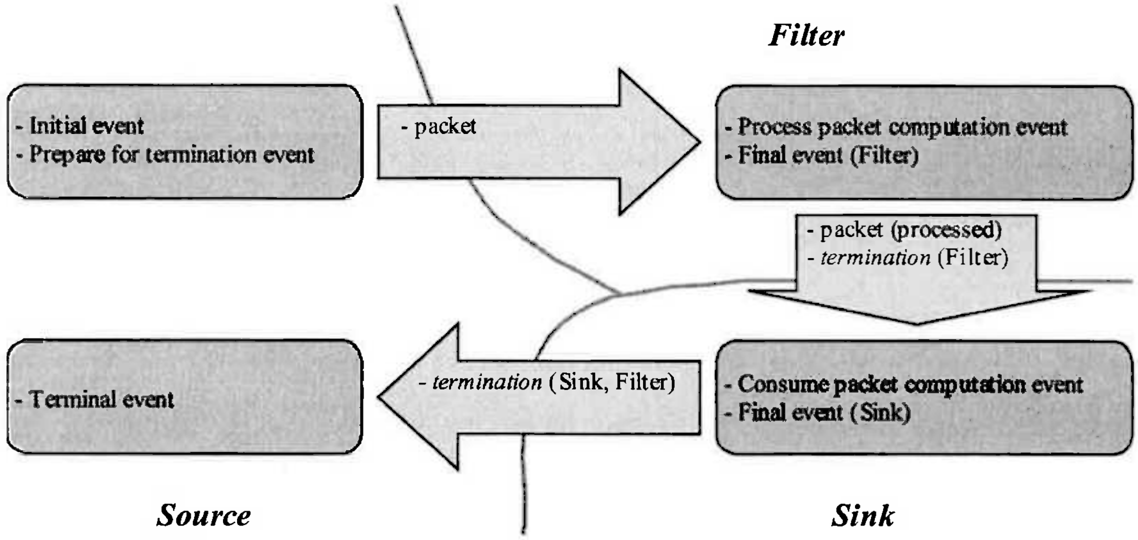

Figure 41 – A Simple Pipeline Interaction.

sends an error message to its immediate upstream neighbour, and so on. There are two merits motivating this approach:

- Error handling is as close to the point of detection as possible. The immediate upstream pipeline element is the first to receive the error indication; if the error can be dealt with at that point, then it is. Otherwise, the error propagates further in the upstream direction.
- Pipeline elements need contain information only about immediate neighbours in the pipeline, which, in practice, they will need anyway (for flow control, for example). If errors were sent to some other location (for example, the pipeline source), then all pipeline members would require information about this source process, which would need to be communicated in some fashion.

In general, filters exhibit one of two responses on reception of a packet:

- Accept and process the packet in some way and send the processed packet to the filter's downstream neighbour.
- Reject the packet and send an error message to the filter's upstream neighbour.

The first is of course the common case.

The filter concept is very closely tied to the use of stream oriented communication. In contrast, communication in object systems is based around discrete operations (method calls). Whilst it is possible to integrate streams into a distributed object system, the best approach to doing so is not immediately obvious. This thesis concentrates exclusively on remote operations communication from this point on, and hence communication between filters is not discussed further.

5.1.3.3 Peers

Peers are the most general type of process in distributed systems. In principle, any process that interacts (via messages) in a way not covered by the categories of client, server, layered server, administrator or filter is a peer.

In practice, the most common types of peer interactions are bilateral "conversations" between two[77] peers. The peer that initiates such a "conversation" is called the *initiator*. The other peer in the conversation is called the *responder*. However, after the conversation has commenced, the two peers

[77] Interactions based on point-to-point message sends involving more than two concurrent peers are certainly possible; however, the semantics of such interactions, particularly in the presence of failures, are extremely complex. In practice, where more than two processes are concurrently involved in communication, one is usually better off using multicast (group communication) primitives. This technology is beyond the scope of this thesis.

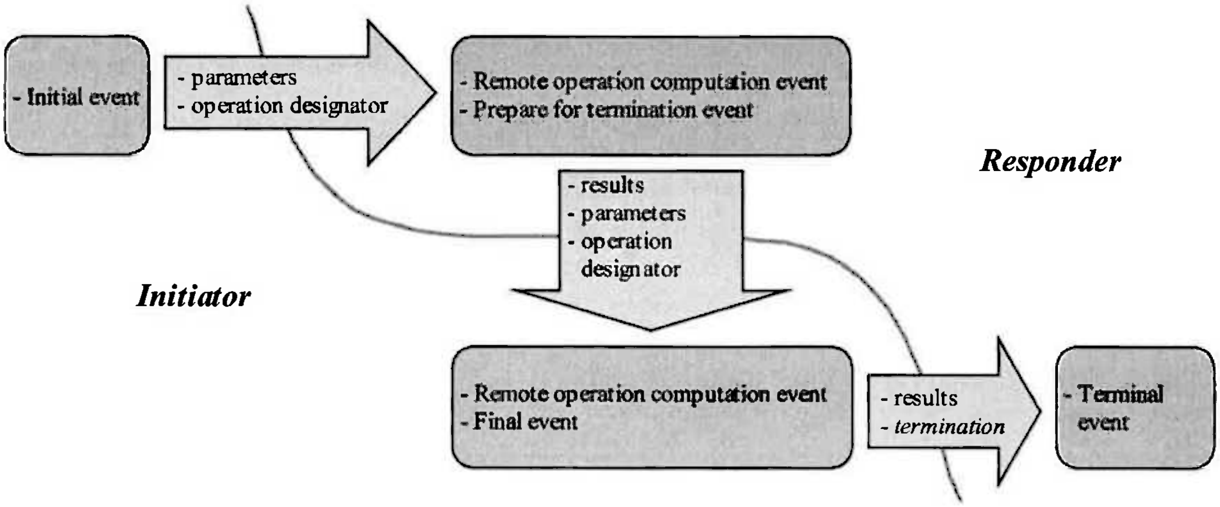

Figure 42 – Simple Peer Interaction.

do not have distinct roles, in contrast to the case in client/server interactions. There are two main differences between two-peer interactions and client/server interactions:

Symmetry – in client/server interactions, the client "drives" the interaction: all computation is caused by the client, either directly (autonomous execution on the client) or indirectly, by sending a request to the server. The server performs computation only in response to client requests. In contrast, two-peer interactions are symmetric: both peers may perform autonomous computation, and both may trigger computation in the other.

Termination – in client/server interactions, the client is the termination coordinator. In a two-peer interactions, either peer might be termination coordinator; depending on the particular interaction.

The most basic two-peer interaction is shown in Figure 42. The responder executes operations in response to a message from the initiator, concluding with a message back to the initiator. The initiator executes operations in response to this message, concluding with a message back to the responder, which receives the message, thereby terminating the interaction. Note that the responder acts as the termination coordinator for the interaction.

5.1.4 Effective Support for Inter Object Communication

The issue of providing effective language support for each significant pattern of communication identified in the previous section is now addressed. In each case a scheme for classifying the level of support provided by a language is developed.

5.1.4.1 Marshalling and Unmarshalling

The task of marshalling and unmarshalling messages is usually addressed below the language level. It may be useful to provide a (reflective) facility for language level access to these operations: this permits specialised marshalling for classes that require it, and may also permit programmers to implement replication for performance critical objects. However, only a small subset of language users would be expected to employ such a facility. Discussion of such reflective facilities, and their role in production (versus experimental) programming languages is beyond the scope of this thesis.

5.1.4.2 Requests

At the client end, language level support for request messages must deal with several issues, as listed below:

Request Target – a means of identifying the target (process) of the request.

Request Operation Designation – a means of identifying the operation requested.

Request Parameter Provision – a means of supplying parameters to the request.

Fortunately, object-based languages intrinsically support these requirements. As seen in Chapter 3, method calls on objects encode three pieces of information independent of language syntax. For example, the syntax used in Eiffel and C++ is as follows:

```
Object.method(parameters)
```

Here *Object* encodes the target object of the request. This is not necessarily identical with the target process, rather the target process is the process containing the target object. In a system supporting replication, there may be several such processes. Nevertheless, the point remains valid: given an object to operate on, it is possible to find the process where that operation is to be effected, hence the first requirement listed above is addressed. The second and third requirements are directly addressed by the *method* and *parameters* entities at the language level.

Language level support for requests at the server end is primarily concerned with provision of facilities for managing concurrent incoming requests (called "request scheduling" in [Papathomas 1992]). The provision of effective support for request scheduling is very important. In general, if the resulting system is to be truly easy to use, this support should include aspects at the language level. Furthermore, because of the inheritance anomaly (see Section 4.1.6.3), the provision of such support has proven intensely challenging. However, this framework is about the assessment of communication, so this issue is not addressed *per se*. On the other hand, it does arise in the assessment of software-engineering support with respect to method pre-conditions because such pre-conditions, which are independently useful in addressing software-engineering concerns, imply constraints on request scheduling.

5.1.4.3 Replies

In order for reply facilities to be provided by a language, that language must incorporate a core notion of reply, being a message in response to a request message. Consequently, there is a basic level of support for the core notion of reply. This is a pre-requisite for all other reply facilities.

Given core support for replies, fours kinds of categories are needed to assess effective use. One of these is a server (reply sender) side category:

Reply Sending – the facilities provided to transmit reply messages.

Two other categories are client (reply receiver) side:

Reply Matching – the facilities provided to match incoming reply messages with previously sent requests.

Reply Synchronisation – the facilities provided to synchronise with the arrival of incoming replies.

The final category concerns the concept of reply itself:

Reply Kinds – the kinds of values that may be sent as replies.

5.1.4.3.1 Reply Sending

In many communication mechanisms, reply sending is associated with remote operation return. This ensures that replies are sent, given that the remote operation returns. In other communication mechanisms, there may be an explicit reply sending operation, in addition to or instead of the return operation. This is termed an *early reply* operation, it differs from return in that it does not cause the remote operation to terminate. Reply sending facilities are assessed by a Boolean value indicating presence of an early reply construct.

5.1.4.3.2 Reply Matching

Reply matching is assessed by classification into levels as follows:

1 – *No support* – there is no system support for the matching of replies to corresponding requests. Any reply matching must therefore be implemented entirely by developers using the system.

2 – *Tagging* – the system automatically marks request messages with unique tags, and similarly marks reply messages with the tag of the corresponding request message. This information may be used by programmers to effect reply matching (via a hash table of outstanding requests, for example).

3 – *Full support* – reply matching is entirely hidden within language constructs. For example, with an RPC-based communication mechanism, the results of incoming replies are returned to the client in the same way as values returned by a local procedure call.

5.1.4.3.3 Reply Synchronisation

This framework identifies three main levels of support for reply synchronisation. The levels are as follows:

1 – *Asynchronous* – replies are delivered via a method call to the client.

2 – *Explicit Synchronous* – the language provides the facilities via explicit operation(s) executed synchronously by the client.

3 – *Implicit* – these facilities are provided in a way that does not required explicit use of a language feature by the programmer, this is generally the case with synchronous RPC.

When viewed in isolation, the implicit level of support offers the simplest (and therefore preferred) alternative, however, other factors may require an explicit approach. It is also possible to provide both, leading to another category, assessed at the same level as the implicit category:

3 – *Hybrid Implicit/Explicit Synchronous* – implicit support is provided for simplicity and in addition this may be bypassed and resort made to explicit support where necessitated by other factors.

5.1.4.3.4 Reply Kinds

There are three general kinds of values in a distributed object system, as follows:

- *Local Reference* – a value which is a reference to an object in the local address space.
- *Remote Reference* – a reference to an object in some remote address space.
- *Non-reference* – a basic value, such as an integer, or a user defined basic value, which is accessed directly, rather than via a reference.

Any combination of these kinds of values might be supported by a reply mechanism, the assessment in this category enumerates the set of supported reply kinds.

5.1.4.4 Coordinated Termination

Coordinated termination is the ability for the termination coordinator for an interaction (usually the client is the coordinator) to determine that all participants in that interaction have terminated. Some communication mechanisms may ensure coordinated termination automatically, whereas others may shift the burden or some part of it onto programmers. The levels for assessment of support for coordinated termination are as follows:

1 – *No support* – the communication mechanism provides no support for coordinated termination, developers are wholly responsible for ensuring it.

2 – *Supported* – the communication mechanism provides support for coordinated termination, developers may use that support to ensure it.

3 – *Automatic* – the communication mechanism automatically ensures coordinated termination in all situations.

5.1.4.5 Reuse of Existing Code

The majority of existing code is both sequential and centralised. A middleware approach that supports the introduction of concurrency and distribution into existing systems whilst limiting the need to modify that code is clearly to be preferred over a middleware system that requires substantial modification of the existing code base. There are two issues to be considered here:

- ***Syntactic Disruption*** – the syntactic changes to existing code that are required by the introduction of the middleware.
- ***Semantic Disruption*** – the impact of the changes in semantics due to distribution of the system on the existing code base.

5.1.4.5.1 Syntax

It may be possible to eliminate any need for syntactic changes to existing programs and have the middleware perform distribution invisibly. However, it is likely to be desirable to exercise some control over distribution, such as whether a given object may be located on a remote node, and for this, some alteration of the source code will be necessary. This aim should be to minimise the number and scope of these changes, noting that remote method call (and return) is by far the most common operation that is related to distribution. The levels of assessment are:

1 – *Significant syntactic disruption* – alterations are necessary to all parts of the program in which operations related to distribution occur. These operations include remote method calls.

2 – *Limited syntactic disruption* – alterations are restricted to specific kinds of distribution related operations, not including remote method calls.

3 – *No syntactic disruption* – code does not need to be altered at all, distribution is performed implicitly by the middleware.

5.1.4.5.2 Semantics

It is infeasible in general for a distributed version of a system to exhibit the same semantics as version of that same system that is not distributed. Given this, the goal should be to produce a middleware system that changes the semantics in a simple and consistent fashion, such that an existing code base continues to function correctly in the distributed context. The levels of assessment are:

1 – *Significant semantic disruption* – the changed semantics mean the existing code base needs to be substantially reworked in order to support distribution.

2 – *Consistent and simple variation in semantics, with exceptions* – in general, existing code continues to function correctly with the changed semantics. In a limited number of specific cases, however, there is a radical change to the semantics that result in incorrect behaviour of existing code.

3 – *Consistent and simple variation in semantics* – existing code continues to function correctly with the changed semantics.

Note that when there is significant syntactic disruption of the existing code base, consideration of the semantic disruption does not arise, since the existing code will have to be changed in any event.

5.1.4.6 Extended Client/Server Interactions

A number of issues arise in the provision of support for extended client/server interactions. These include:

One way Requests – the ability to send requests for which there is no corresponding reply, not even an empty reply.

Layered Servers – support for layered server architectures.

Request Forwarding – support for forwarding a request between servers.

5.1.4.6.1 One-way Requests

Recall that in the model of interactions exhibiting coordinated termination, one way requests are meaningful only within the context of some larger interaction. Specifically, if a client is involved in an extended interaction with a server, it can use one way requests within that interaction, provided that the last request is acknowledged and the server executes requests from a given client in the order sent. Specific support for one-way requests allows the client to continue computation immediately after submitting each such request. This support is assessed by a Boolean value indicating its presence.

5.1.4.6.2 Layered Servers

In order to support layered servers effectively, it is necessary to consider the issues raised in [Liskov, Herlihy and Gilbert 1986]. Essentially the problem is as follows: if a server has both static process structure and uses synchronous communication, the situation can arise where all processes in the server are blocked calling lower layer servers. If this happens, servers in the layer immediately above cannot call operations of the server, even though it is not performing any computation. Thus, the progress of the operations being executed by the higher level servers is stalled purely for structural reasons. These aspects are primarily performance related, hence they are considered in the performance component this framework (see Section 5.3).

5.1.4.6.3 Request Forwarding

Administrators can be viewed as layered servers. However, it is possible to take advantage of a characteristic of administrators: namely, that having selected an appropriate server and sent the client's request to it, the administrator really has no further involvement in the interaction. If a layered server implements the administrator, it will of course receive a reply from the server it sent the request to, and will then send that reply back to the client. However, there is no real reason for the reply to go via the administrator in this way. An alternative is the provision of a special primitive to enable the reply to go directly to the client.

This capability is termed *request forwarding*, it provides the ability for the implementation of a remote operation, R, to make a request to a remote operation, R', in another process instead of sending a reply. The reply for R' is then directed to the destination expecting the reply to R and substitutes for the reply to R.

The interaction graph for a request/reply interaction mediated by a single administrator using request forwarding is shown in Figure 43. As can be seen in the figure, the administrator's role in the interaction is finalised when it sends a request (with forwarding) to the server. The administrator's termination indication is delivered to the termination coordinator (the client) via the server.

This clearly imposes constraints on the replies: they must either both be void or must contain type compatible results. However, there are not similar constraints on the requests. Finally, it should be noted that recursive request forwarding is permitted; there is no limitation to the case with one administrator given above. Hence, hierarchies of administrators can be supported. Request forwarding may be supported at four levels:

1 – *No Support* – there is no ability to do request forwarding.

2 – *Support via Reflection* – ability to do request forwarding by manipulation of message data structures via reflection.

3 – *Linguistic Support* – provision of a language construct to effect request forwarding.

4 – *Automatic Support* – the language compiler is able to find and utilise possibilities for request forwarding without programmer intervention.

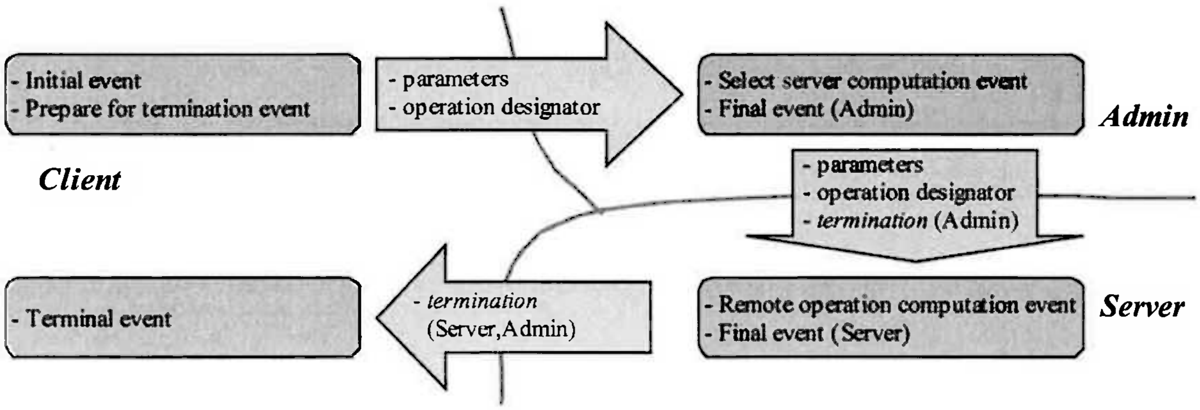

Figure 43 – A Remote Request Dispatched by an Administrator.

Each level listed above is an improvement on its predecessor in terms of either ease-of-use or effectiveness or both.

5.1.4.7 Peer Symmetry

In contrast to client/server interactions, peer interactions exhibit symmetry in the roles of participants. In particular, any of the peers in an interaction might pro-actively generate a new message: no single peer is responsible for driving the interaction. Support for symmetry of peers is assessed by a Boolean value indicating whether symmetry is supported, as opposed to programmer being forced to embed peer interactions in client/server interactions.

5.1.4.8 Generalised Reply Entailment

In a simple client/server system, the sending of a request by a client to a server entails the server send a reply to that client. In general, any message in an interaction can entail a reply. This is in contrast to there being a strict division into requests and replies. In particular, a message that is a reply for some previous request may also serve as a request entailing a subsequent reply. There are four areas in which generalised reply entailment is assessed:

Reply Entailment – the overall level of support for reply entailment.

Reply Races – support for races in which several remote operations compute a given result in parallel.

Multiple Reply Entailment – the ability for a given message to entail more than one reply.

Interaction Structure – the flexibility provided in the structure of an interaction consisting of an initial request and subsequent entailed replies.

5.1.4.8.1 Reply Entailment

The general level of support for reply entailment can be classified as follows:

1 – *No support* – there is no support whatsoever for replies.

2 – *Simple* – only the simple client/server form of reply is supported.

3 – *Extended* – any message, including a reply message, can entail a reply.

5.1.4.8.2 Reply Races

Suppose one needs to obtain a result of a certain parameterised computation in the shortest possible time, in the following circumstances:

- A range of servers is available, but the time taken by each to compute the result fluctuates due to variations in load.

- Different servers use different algorithm to compute the result. The fastest algorithm for the computation is determined by the parameters, but selection of that algorithm requires time comparable to the time taken to perform the computation.

One approach to this problem is to submit requests for the computation to several servers simultaneously. This establishes a race between each of these servers to compute and send the result. These requests could be completely independent, producing independent results, in which case the application programmer must in some way identify and synchronise with the earliest arriving result. A better alternative is for the communication mechanism to support reply races, by allowing several request to be responsible for producing a given result. The application programmer then synchronises with the availability of that (single) result.

In the situation described above, only one reply (the first to arrive) is of interest, because the values of all the result are the same. In some situations, the values of the various replies may differ, and therefore several, possibly all, values are of interest. For example, some of the servers might implement quick but only moderately accurate, algorithms, whereas other servers might implement slower algorithms with more accuracy, and the performance requirement might be for the most accurate result within a given deadline. System support for this situation would make all the reply values available.

In summary, the level of system support for reply races can be categorised according to the following levels:

1 – *No support* – no support for reply races.

2 – *Winner result only* – only the winner value for the result is available.

3 – *All results* – all result values are available.

5.1.4.8.3 Multiple Reply Entailment

In a simple client/server system, each request entails a single corresponding reply. It is also possible to provide support for a request (or more generally, any message entailing a reply) to entail more than one reply. This facility is assessed by a Boolean value indicating presence of such support.

5.1.4.8.4 Interaction Structure

In a system supporting reply entailment to the extended level, request/reply message sequence can include more than the two messages in the request/reply pair of simple client/server systems. Each of these messages has a type determined by the types of the values it contains.

For request/reply sequences including three or more messages, the communication mechanism may provide some flexibility in the sequence of message types for a given request/reply sequence. The degree of flexibility can be classified as follows:

1 – *Fixed* – the sequence of message types is fixed.

2 – *Regular* – the sequence of message types can be defined by a regular expression.

3 – *Recursive* – the sequence of message types can be defined by a recursive grammar formalism, such as BNF.

4 – *General* – the sequence of message types is more general than can be defined by a recursive grammar.

The first message in any request/reply sequence is of a fixed type by definition, because it initiates the sequence. Furthermore, the second message in a request/reply sequence is fixed in typical client/server systems, because the type of that message is part of the type (the result type) of the first (request) message.

5.1.4.9 Linguistic Heterogeneity

As described in Section 2.4.1.3, heterogeneity of all kinds, including linguistic heterogeneity, is an unavoidable characteristic of realistic distributed systems. Middleware must support all kinds of heterogeneity, but support for architectural and operating system heterogeneity is mostly a matter of porting the implementation to each architecture or operating system in the system. Consequently, linguistic heterogeneity is the primary challenge.

Middleware that supports linguistic heterogeneity provides access to equivalent functionality from all programming languages used to implement computational elements of the system. Consider the set of languages used to implement new computational elements of the system (as opposed to legacy code). This set of languages is the user base of the middleware facilities. Linguistically heterogeneous middleware will provide each language in this set with complete access to all the middleware facilities. This principle of equality forms the basis of the definition of linguistic heterogeneity used in this thesis.

There is a degenerate case that satisfies the principle of equality as given above. If there is only one language used to implement the computational elements of the system, then there is, by definition, equality of access to the middleware. However, this case is clearly not linguistically heterogeneous, so a refined principle is needed: that there be two or more different languages provided with equality of access to the middleware.

In this framework, middleware is assessed in respect of linguistic heterogeneity by a Boolean value indicating whether the middleware can be implemented to be compliant with the refined equality principle.

5.2 Analysis of Support for Software Engineering

The past two decades have seen a major change from *programming*, with the emphasis on writing code to *software engineering*, which attends to all stages of the software life cycle. Object-based (and object-oriented) software engineering techniques are a relatively recent addition to the software engineer's armamentarium. However, previous chapters of this thesis have noted the close compatibility between object technology and distributed systems communicating via explicit message passing. Consequently, this thesis analyses the support for software engineering within an object-oriented context.

The software engineering component of the framework for analysis contains two parts, these are as follows:

1. Development of a framework for the assessment of support for contract-based software engineering.
2. Development of a set of constraints on the mechanisms admissible for assessment within the overall framework is applicable, defined in terms of software engineering issues.

5.2.1 Contracting Support

As discussed in detail in Chapter 3, the notion of contract is a fundamental aspect of software engineering using object technology. This section is arranged in three parts:

1. Identification of desirable principles for contract specifications considered in the abstract, in Section 5.2.1.1.
2. The application of the abstract principles to concrete language constructs, specifically, the type system (Section 5.2.1.2), the assertions system (Section 5.2.1.3) and communication structure, in terms of reply entailment and reply validity (Section 5.2.1.4).
3. Consideration of support for exception handling (Section 5.2.1.5)

5.2.1.1 Abstract Principles for Contract Specifications

This subsection identifies core characteristics of language support for contracts. The aim is to develop these principles for contract support in the abstract, which are then used in later subsections to evaluate concrete language facilities.

5.2.1.1.1 Coverage of Contracts

Ideally, contracts should cover all the obligations and all the benefits for each of the contracting parties. The effectiveness of language support for contracting is largely determined by the degree to which the breadth of coverage approaches this ideal.

5.2.1.1.2 Explicit Contract Specification

There is always an implicit specification of the contract associated with a software entity. This is defined by the way the entity functions in response to the way it is actually used by the other entities in the system. However, the disadvantage of implicit specification is that the specification may be other than that intended by the programmer. In particular, it may be incorrect with respect to system requirements. Consequently, only explicit contract specification capabilities are assessed in this framework, implicit specification capabilities are ignored.

5.2.1.1.3 System Responsibility for Contract Enforcement

As much as is possible, the language should relieve programmers of the responsibility for enforcing contracts. Contract enforcement is repetitive during development, and therefore tedious. It is consequently error-prone if done by developers. Automatic checking, of course, has no such problem, it can safely relieve developers of much of the burden of contract enforcement, freeing them to concentrate on specification and implementation of contracts.

This framework is concerned with assessment of language level support for communication. Consequently, the assessment excludes facilities for contract specification which are not enforced by the system.

5.2.1.1.4 Early Contract Enforcement

Contract violations indicate defects in the software or inadequacies in the environment of its execution. It is desirable that these defects should be detected as early as possible, preferably at compile time. Therefore, contract enforcement should be as early as possible, in particular, checking should be static wherever possible.

Furthermore, if dynamic checking is required, it is desirable that an executing program unit should reach a state in which it has passed all run-time checks required to ensure its validity. That is, dynamic checks should be *clustered* at the beginning of execution of a program unit.

5.2.1.1.5 Declarative Contract Specification

Operational specifications describe the properties of a contracting entity in terms of an idealised implementation realising those properties. Whilst this implementation might be high-level and abstract, it is still an implementation. Mistakes in the operational specification are likely to propagate to the (real) implementation, simply because of the ease of translating the operational specification into an actual implementation.

In contrast, declarative specifications are more minimal descriptions of the desired properties and are in no sense implementations. The production of an implementation of this kind of specification is more rigorous and challenging, and, in being so, less likely to propagate errors in the specification. Furthermore, because the specification is more concise, it is less likely to contain errors. Consequently, the mode of specification should favour declarative expression over operational expression.

5.2.1.1.6 Positive Contract Specification

It is better to accidentally exclude correct, but marginal, cases from a contract than to accidentally include cases where an entity fails to operate correctly. Even a casual study of the history of human endeavour indicates that sins of omission are more common than sins of commission. That is, it is more commonly the case that some necessary action is omitted, than that some wrong action is committed. If contract specification is positive, specifying permitted cases, an omission will result in a contract that is overly restrictive, that delivers more than it promises. On the other hand, if specification is negative, an omission will result in a lax contract, that fails to deliver all that it promises. The result of the former omission is an additional burden on the users of the entity to which the contract is attached, and an unnecessary restriction on the scope for re-use of that entity. This is inefficient and undesirable, but does not effect system correctness. The result of the latter omission is a contract violation and system failure, this is clearly the less desirable of the two alternatives, and therefore contract specification should be positive rather than negative.

If a specification is given implicitly, then that specification is positive by definition. Consequently, it is only meaningful to assess application of the positive specification principle where specification is also explicit.

5.2.1.1.7 Intrinsic Contract Specification

In object technology, classes are the primary elements of re-use. To support re-use at the module (class) level, contracts should be intrinsic to each module, rather than extrinsic and applying to a collection of modules.

Obviously, there is a problem where several distinct entities are jointly subject to a contract specifying properties of the group as a whole. An example of this is seen in Chapter 3. Such situations certainly arise in practise, consequently intrinsic specification cannot be universal, but where it is sufficient, it should be used. In other words, the specification technology should not force developers to use extrinsic specifications where it is not required by the software under development.

5.2.1.1.8 Inseparable Contract Specification

The set of contracts associated with a program module describes its relation to its environment. A module can only be correctly re-used in an environment that is compatible with this set of contracts. To assist enforcement of correct re-use, a contract specification and the module implementing that contract should be inseparable.

Intrinsic contract specification implies inseparable contract specification, but not the reverse. Consequently, compliance with the inseparable specification principle only needs to be assessed if specification is extrinsic.

5.2.1.2 The Type System

The type system is an important subset of contract specification support. This section applies the abstract principles for contract support to assess the type system support provided by middleware mechanisms.

5.2.1.2.1 Coverage of the Type System

Contract coverage applied to the particular case of the type system corresponds to richness and modelling capability exhibited by that type system. In a middleware specification such as CORBA [OMG 1997], this corresponds to the expressiveness of the IDL of the middleware. The expressiveness of the IDL is clearly an important determinant of the effectiveness of the middleware defining that IDL. However, in order to be useful in a linguistically heterogeneous environment, the IDL must support straightforward mappings (often called bindings) to and from common programming languages. More specifically, the mappings are defined between the IDL and the type systems of those programming languages. In this context, there is a tendency for the IDL design to follow those type systems as closely as possible. Consequently, IDL design, (and hence IDL expressiveness) is largely determined by factors independent of communication requirements. There is therefore no application of the coverage principle to the assessment of the type system.

5.2.1.2.2 Early Type Checking

Static type checking prevents type errors at compile-time, prior to the execution of the system. Consequently, it is to be preferred over dynamic type checking in all cases where it is possible. However, as seen in the previous chapter, distributed systems require at least some dynamic type checking. The kind of type checking can be classified as follows:

1 – *Dynamic* – all type checking is performed at run-time.

2 – *Dynamic and static* – type checking is performed statically only in the majority of cases where implementation is straightforward, and is dynamic otherwise.

3 – *Static with exceptions* – type checking is performed dynamically only where it is impossible to do the checking statically[78].

Once again, these are in order of increasing desirability. The desirability of static type checking is a direct application of the principle enunciated in Section 5.2.1.1.7: that contact enforcement should be as early as possible.

As mentioned in the previous chapter, some measure of dynamic type checking is unavoidable in distributed systems. In addition to minimising the number of dynamic checks which must take place, it is also desirable to cluster those checks as much as possible. In particular, it is desirable that checks take place early in execution. Furthermore, if possible, a given instance of a system module should reach a state in which all possible dynamic checks in that instance have occurred and consequently that instance is known to free from type errors from that point forward. However, clustering of dynamic type checks is largely a property of the languages used in the implementation of system elements, rather than of the middleware. Therefore, the property to assess is the compatibility of the middleware with clustering in implementation languages. This is assessed by a Boolean value indicating whether the communication mechanism is compatible with clustering of dynamic type checks. One example of how clustering of dynamic type checks associated with run-time binding might be promoted by a language mechanism is given in the previous chapter.

5.2.1.2.3 Declarative Type Specification

Type information, when given explicitly in program texts, is given in a declarative rather than an operational form. Although it is possible to conceive of an operational approach to explicit type specification, the designs of mainstream programming languages have adopted a declarative approach. Consequently, it is superfluous to apply the declarative mode principle to type specification for communication middleware.

5.2.1.2.4 Positive Type Specification

Type information is typically given in a positive form rather than in a negative form. Interface specifications list the valid operations available on a module and the valid types of arguments to those operations, rather than listing invalid operations and arguments. It is possible to conceive of designs where type specification is negative, but the absence of mainstream examples makes consideration of this alternative pointless. Consequently, it is superfluous to apply the positive specification principle to type specification for communication middleware.

[78] This level includes the case where there are no exceptions, since, by definition, this lack of exception must arise form some limitation in the language design. The position taken in this thesis assumes that distributed systems require some dynamic type checking, so this point is moot.

5.2.1.2.5 Intrinsic Type Specification

In main stream programming languages, explicit type information is intrinsic to modules rather than extrinsic. It is possible to conceive of designs where type specification is extrinsic, but intrinsic specification is ubiquitous in real type systems. Consequently, it is superfluous to apply the intrinsic specification principle to type specification for communication middleware, since specification is always intrinsic.

5.2.1.3 Assertion Checking

Assertion checking is another important subset of contract specification support. Support for assertions in current programming languages is uncommon. However, it is difficult to see how a language or mechanism could be incompatible with the addition of such a facility, so there is no assessment of whether assertion checking is supported. Of course, assessment of the details of assertion checking, as described below, is of course contingent on there being such support, and is not applicable in its absence.

Given that the middleware (communication mechanism) provides support for checking assertions, it should deal with issues raised by supporting assertions in the sequential context. This is assessed done by application of the abstract contracting principles to analysis of the assertion support.

In addition, the introduction of asynchrony raises complications in assertion support. These are as follows:

- ***Precondition Paradox*** – the need for dual semantics of preconditions in asynchronous environments.
- ***Asynchronous Contract Completion Anomalies*** – discussion of the complications by asynchronous fulfilment of contracts.

These are discussed in Sections 5.2.1.3.7 and 5.2.1.3.8 after application of the principle to the synchronous case.

5.2.1.3.1 Coverage of Assertion Specifications

Assertions are based on a theoretical underpinning in mathematical logic. The simplest form of mathematical logic, termed propositional calculus, consists of simple Boolean expressions. This is extended with quantifiers and logic variable to form predicate calculus. This second form of logic is able to support reasoning about sets of values.

From this theoretical background, linguistic support for coverage of assertion specification may be assessed as follows:

1 – *Propositional* – only simple Boolean expression can be specified.

2 – *Predicate* – assertions on sets of values can be specified.

It is sometimes necessary to maintain properties on several objects as a group, so there is an additional level:

3 – *Predicate Multi-object* – assertions can apply to relationships between objects, rather than just properties within a single object.

5.2.1.3.2 Intrinsic Assertion Specification

Assertions that apply to properties of a single object (including objects attached to its instance variables) are naturally given in a form that is intrinsic to the object (or more precisely, the object as an instance of its class or type). It is difficult to see why a language design including assertion support would go against this natural form in supporting single object assertion specifications.

Conversely, assertions that apply to several related objects that not part of a common containing object are not only naturally extrinsic, but actually are extrinsic by necessity[79]. A language design to support such multi-object assertions would need to include a means for extrinsic specification. The designer of such a language might be tempted, in the pursuit of apparent simplicity, to support only extrinsic specification covering both the single object and multi-object cases. This would be a mistake, as it would unnecessarily expose the single object cases to the problems associated with extrinsic specification.

Intrinsic assertion specification is assessed by a Boolean value indicating whether assertion specification is intrinsic wherever it is possible to be so.

5.2.1.3.3 Inseparable Assertion Specifications

As described previously, assertions may be extrinsic of necessity in some cases. This admits the possibility that the specification is also separable, which is neither necessary nor desirable. The application of the principle of inseparable specifications is assessed as follows:

1 – *Separable* – specification is separable in all cases (this implies that it is also extrinsic).

2 – *Partially separable* – specification is separable in cases where it is extrinsic.

3 – *Inseparable* – specification is inseparable in all cases.

5.2.1.3.4 Declarative Assertion Specification

The mathematical expression of both propositional calculus and predicate calculus is declarative. Implementation of checking of assertions expressed in propositional calculus is straightforward. However, implementation of checking of predicate calculus expression is both non-trivial and inefficient. Therefore, languages incorporating assertion support, such as Eiffel, typically support only propositional terms in declarative form. Support for predicate calculus assertions is provided by allowing these propositional terms to include function calls. These functions can then evaluate any expression whatsoever, but are operational in form. Support for declarative assertion specification is assessed at the following levels:

1 – *Operational via Functions* – assertions are specified by the association of precondition and postcondition Boolean functions with methods and invariant Boolean functions with classes. The specification is thus wholly operational.

2 – *Declarative Propositional Terms with Functions* – only simple Boolean expressions are specified declaratively, other expressions are specified operationally.

3 – *Declarative Predicate Terms* – all assertions are specified declaratively.

5.2.1.3.5 Positive Assertion Specification

Calling for assertions to be positive specifications is redundant! It is in the nature of assertions to be positive. Whilst it is possible to use logical negation to make negative assertions, this is a choice that the developer makes, it is not required by the language system. Therefore, assessment of the application of this principle is superfluous.

5.2.1.3.6 Early Assertion Checking

Static checking of assertions is, in general, still at the research prototype stage, although much progress has been made, see [Detlefs 1995] for an example. Consequently, one would expect, at best, a combination of dynamic and static checking. Application of the early checking principle can be assessed as follows:

1 – *Wholly dynamic checking* – all assertion checking is dynamic.

[79] There is a degenerate case, wherein the multiple objects involved in an assertion are of the same class. In this case, multi-object assertions can (and should) be intrinsic to the class of the objects.

2 – *Partially static checking* – significant aspects of assertion specification are checked statically.

5.2.1.3.7 The Precondition Paradox

Recall from the previous chapter that pre-conditions involving separate objects must operate as synchronisation constraints rather than correctness constraints. Meyer terms this the 'precondition paradox'.

Meyer's solution to the problem is to delay execution of any client routine making calls to separate objects until all precondition of that routine involving separate objects are true. In other words, precondition clauses involving separate objects cause the client to wait to be synchronised with the remote objects.

This works well in many situations, and is the only solution when a client must have exclusive access to a remote object or objects within a critical section. However, as noted in the previous chapter, there is another alternative appropriate to some situations, which is to enqueue a remote operation at the server, until that operation's preconditions are fulfilled. This provides an extra choice for synchronisation, but should be viewed as augmenting, rather than replacing, Meyer's solution. The way in which a middleware system deals with the precondition paradox is assessed by classification as follows:

1 – *No synchronisation semantics* – preconditions lack any synchronisation semantics.

2 – *Reflection* – preconditions lack any synchronisation semantics, but there is reflective access to requests, enabling server side queuing to be supported by the (server) class programmer.

3 – *Server queueing* – requests queue at the server end until their preconditions are met.

4 – *Client synchronisation* – client routine preconditions have the semantics espoused by Meyer [Meyer 1997][80], and discussed earlier in this thesis (Section 4.1.5.1)

5 – *Client synchronisation and server queueing* – in addition to client synchronisation, server side queueing is available.

5.2.1.3.8 Asynchronous Contract Completion Anomalies

As explained in Section 4.1.7.2, asynchrony in remote invocations leads to a requirement for a client to be able to synchronise with the fulfilment of contracts associated with those remote method invocations. Middleware support is assessed as follows:

1 – *No support* – no direct support, the client must obtain contract completion information as a side effect of an otherwise superfluous function call on the remote object.

2 – *Exception on failure* – no direct support for synchronisation, but the client is notified by an exception if the remote contract could not be fulfilled. Exception handling is described later.

3 – *Notification on completion* – a method of the client is called (by the middleware) when the contract is fulfilled.

4 – *Optional synchronisation* – an operation is provided enabling the client to wait for the completion of one or more contracts that it has initiated.

Some communication mechanisms may exclude the possibility of asynchronous contract completion, in which case this assessment category is not applicable.

5.2.1.4 Contracting for Replies

In abstract terms, there is a *reply entailment* contract between a *reply sender* and a *reply target* as follows:

- The reply sender is obliged to send a reply for any reply entailment contracts it has accepted.

[80] Note that this is contingent on there being a mechanism for reservation of exclusive access to separate objects in preconditions.

- The reply target is entitled to expect a reply for any reply entailment contracts it has initiated and that are accepted by the reply sender.

In simple client/server systems, the server is the reply sender and the client the reply target. Contract instances are initiated by the client calling a remote operation that has a reply entailment contract. Acceptance of a reply entailment contract by the reply target might be dependent on other contract clauses, for example it might depend on the truth of preconditions.

There is also a *reply validity* contract between the reply sender and reply target. This has the following clauses:

- There is a restriction on the replies a given reply sender may send to a given reply target. The sender may only send replies specified in contracts initiated by the target.
- The target is obliged to accept all replies specified in contracts that it initiated.
- The target need not handle replies other than those specified in contracts that it initiated.

Any reply validity contract is coincident with a reply entailment contract, and consequently is initiated in the same way (by the client in the simple client/server case. In addition to the reply entailment and reply validity contract pair, there may be other contract clauses that are relevant, for example, an assertion clause might constrain validity of result values included in replies.

In simple client/server systems, reply entailment and validity specifications are given in each server module, in the following forms:

- Functions in the server interface indicate the type of the value sent in the reply. The reply entailment contract associated with a function obligates the server to send a reply and entitles the client to expect that reply. The corresponding reply validity contract entitles the server to send the reply and obligates the client to accept it.
- Procedures in the server interface that are not marked as one-way indicate entailment and validity of a reply without any result values, but are otherwise the same as functions.
- Procedures in the server interface that are marked as one-way indicate the absence of reply entailment and validity contracts associated with that operation.

Reply validity specifications are automatically enforced by the structure of RPC communication in simple client/server systems. This form of reply entailment specification is ubiquitous in extant middleware systems, though many such systems may exclude one-way requests and consequently specification thereof.

In simple client/server systems, reply entailment and validity contracts are essentially trivial and are derived from the structure mapping remote operation return values onto reply messages and therafter to values returned to the client. When these restrictions are relaxed, through the introduction of peer communication and extended client/server interactions, the trivial view no longer suffices, and more powerful means for reply entailment specification must be introduced. Support for reply entailment and validity specification is assessed by application of the contract specification principles. In consequence of the coincidence of entailment and validity contracts, a single assessment may in most case apply to both.

5.2.1.4.1 Coverage of Reply Specifications

Middleware support for reply entailment is discussed previously in Section 5.1.4.8. Support for this facility leads to (at least) an implicit specification of the reply entailment and validity in an interaction. To support contracting, this specification must be made explicit. Furthermore, the coincidence of reply validity contracts with reply entailment contracts is such that an explicit reply entailment contract is also an explicit reply validity contract. Therefore, the assessment of the coverage of reply contracts repeats the levels of Sections 5.1.4.8.1, 5.1.4.8.3 and 5.1.4.8.4, with the assessment indicating the level of explicit specification. The assessment in each case is bounded by the assessment in the corresponding application case. Furthermore, for the majority of mechanisms, the assessed level is the same in the corresponding application and software engineering categories.

5.2.1.4.2 Intrinsic Reply Specification

In procedure call contracts, the procedure is a supplier of some service, and the caller is a consumer of that server. Specification of this contract is intrinsic to the procedure. Reply contracts are analogous to procedural contracts for services, in this case the server being the delivery of a reply. Consequently, specification of reply contracts should be intrinsic to the entity with responsibility for supplying the reply, rather than extrinsically located in some other entity in the environment where the responsible entity is used. Specification capability is assessed by a Boolean value indicating whether the specification is intrinsic.

5.2.1.4.3 Inseparable Specification of Reply Contracts

Reply entailment specifications should also be inseparable from the entity to which they apply. This is assessed by a Boolean value indicating whether the specification is inseparable. Note that intrinsic specifications are always inseparable.

5.2.1.4.4 Declarative Specification of Reply Contracts

The form given above for reply entailment specifications in simple client/server systems is declarative in nature. It is assumed that historical inertia will ensure that such specifications continue to be so for simple client/server interactions. Specification of reply sender obligations in more complex interactions should follow the simple (server) case in being declarative. The assessment is a Boolean value indicating whether the specification of extended reply contracts is declarative.

5.2.1.4.5 Positive Specification of Reply Contracts

In the outline of reply entailment contracts for simple client/server system given above, only the first form (value returning operations) within that outline is obviously positive. The second (non value returning operations) and the third (one way operations) forms are apparently negative. These forms can in fact be viewed as positive, when it is realised that a void reply is the common case, marking a procedure as one-way becomes a positive assertion about the properties of that procedure. In addition, even the apparent variation from the principle of positive specification is superficial, because there is a trivial transformation to and from an equivalent positive specification. This transformation is effected by marking all non one-way procedures in some way (perhaps by specifying a void return value, as indeed is done in C, C++ and Java), and removing the marking of one-way procedures[81]. In any case, the inertia of historical practise will ensure that simple client/server interfaces are specified in the way shown above in the future. Consequently, application of the positive specification principle is assessed only in relation to reply contracts extended beyond the simple client/server case. The assessment is a Boolean value indicating whether the specification of extended reply contracts is positive.

5.2.1.4.6 System Enforcement of Reply Entailment Specifications

Enforcement of reply entailment contracts involves ensuring that replies that are contracted for are eventually transmitted. Automatic enforcement of this property is impossible in many theoretical and practical situations, because ensuring the reply is sent involves ensuring that the execution of the remote operation responsible for the reply terminates (the halting problem). In any case, this involves the executable code of the remote operation, which in a heterogeneous system is outside the scope of the middleware, and therefore not considered in this framework. Furthermore, because of the at-most once semantics of message transmission, the middleware can not even ensure that all replies that are

[81] This would have unpleasant side-effect of making one-way requests the default, consequently it should not be adopted in practice.

sent are actually received. The only property in relation that can be reliably assessed is that a remote operation does not terminate without taking care of its responsibilities in respect of sending replies to fulfil reply entailment contracts.

Ensuring that reply sending responsibilities are fulfilled, given that the contracted supplier (a method invocation) terminates, may be supported as either dynamically or (in some cases) statically. However, static enforcement must be done by the programming language used to implement the operation, so the timing of enforcement of this property is independent of the middleware, and assessment is therefore superfluous in a linguistically heterogeneous system.

5.2.1.4.7 System Enforcement of Reply Validity

Enforcement of reply validity clauses in reply contracts may involve the following:

- Ensuring that objects do not receive replies they have not contracted to receive.
- Indicating an error when objects send a reply they have not contracted to send.

It is not clear that the second point involves a contract violation. It might be considered analogous to a vendor who adds a complementary (free), item in addition to items ordered by a customer, which is not generally a violation of the contract between vendor and customer. However, it is important that objects not have to deal with replies that are unexpected. Therefore, support for enforcement of contractual restrictions on replies can be assessed as follows:

1 – *No enforcement* – objects may be subject to the reception of unsolicited replies.

2 – *Invalid reply filtering* – invalid replies are not received are discarded by the system.

3 – *Invalid reply rejection* – invalid replies are forcefully rejected (with an error indication) at the attempt to send.

4 – *Invalid reply prevention* – it is not possible to send invalid replies.

Furthermore, there are two kinds of invalid replies, as follows:

- *Illegal replies* – those sent by nodes that are not entitled to do so.
- *Extra replies* – those sent by nodes that are entitled to send a limited number (usually one) of replies to a given request, over and above the limit.

The framework assesses the enforcement of restrictions on each of these kinds of invalid reply according to the levels given previously.

5.2.1.4.8 Early Enforcement of Reply Validity

For most of the levels of reply validity enforcement in the previous section the timing of enforcement is implicit. Filtering guarantees statically that invalid replies are not received. Prevention guarantees statically that invalid replies are not sent (and hence not received). Rejection guarantees statically that invalid replies are not received but there is a question of timing of the rejection of invalid replies, leading to the following levels of assessment:

1 – *Dynamic reply rejection* – attempts to send invalid replies are rejected at run-time.

2 – *Static reply rejection* – attempts to send invalid replies are rejected by the compiler.

The assessment category applies separately to the two kinds of invalid replies, but is only applicable to those mechanisms assessed at the invalid reply rejection level for reply validity enforcement.

5.2.1.5 Exception Handling

To support software engineering effectively, it needs to be possible to integrate exception handling with the middleware (communication mechanism). In sequential systems, using procedure call as the communication mechanism, the facility with which exception handling may be integrated is beyond doubt. Distributed (and concurrent) systems introduce asynchrony, which militates against straightforward introduction of exception handling, whilst simultaneously making it all the more essential to support exceptions, because of the reliability characteristics of distributed systems.

The concern in this framework is the support for the delivery of exceptions resulting from asynchronous remote method calls. Assessment of exception handling support for asynchronous remote method calls involves the following:

- ***Asynchronous Method Call Exception Handling*** – a Boolean value indicating whether exceptions raised in asynchronous remote method calls can be handled.
- ***Exception Handling Context*** – the depth of context within which an exception raised in an asynchronous remote method call is handled, discussed in Section 5.2.1.5.1.
- ***Exception Handling Flexibility*** – enhanced exception handling context leads to increasingly severe synchronisation. It is desirable to permit the programmer the flexibility to trade-off decreased context in order to obtain reduced synchronisation. The framework assesses this flexibility with a Boolean value indicating whether two or more choices are available with regard to context and synchronisation of exception handling.

Note that because of the relationship between exceptions and (failed) contracts, this topic is related to asynchronous contract completion, which was discussed in Section 5.2.1.3.8.

5.2.1.5.1 Exception Handling Context

Exceptions raised by synchronous method calls are handled in the context of the caller. Exception handling code therefore has access to information such as the values of local variables, which may be used to guide the handling process. This section describes the levels of assessment for the context within which exceptions raised in asynchronous remote method calls are handled:

1 – *Object* – exceptions from asynchronous method calls are propagated to an exception handler method (or special language clause) associated with the caller object. This is defined at the class level, it runs only in the context of the object which made the call, rather than of the (local) method activation which executed the call. A variation on this approach allows an object other than the caller to be specified as the target of the call.

2 – *Reply Based* – this approach to exception handling assumes that there is a mechanism for synchronisation with replies that arrive asynchronously. If a call generates an exception, then that exception is raised at the point(s) at which synchronisation with the arrival reply occurs. The exception is thus raised in the context of the reply to the original request.

3 – *Caller Frame* – an exception is handled by a block of code lexically nested within the caller of the method that raised the exception. This exception handler block has access to the activation record of the caller, from which it obtains context necessary to handle the exception. Note that this requires retention of the activation records of blocks containing exception handlers, which may in turn require heap based allocation of activation records.

4 – *Caller* – exceptions asynchronous method calls are propagated to the (local) method activation that submitted the call that caused them. The termination of this local activation is delayed until it is determined whether any asynchronous calls raise exceptions.

These are listed in order of increasing preference.

5.2.2 Constraints

Previous discussion (Section 2.3.1.5.6) has demonstrated a variety of problems if multi-threading is misused to provide asynchronous remote calls. In addition, Section 4.1.3 established the incompatibility of concurrent execution of operations on object with the contract-based approach to software construction. Consequently, this framework treats communication mechanisms using this form of multi-threading as inadmissible on software engineering grounds. However, these mechanisms may be analysed with respect to the performance and application parts of the framework.

5.3 Modelling Performance

The performance of a software system is measured by the delivery of services to users. Distributed systems are no exception. It is useful to consider the demands for performance from users of a distributed system at two levels:

- ***The Individual Level*** – whether the performance demands of a single user of the system are met.
- ***The Collective Level*** – distributed systems are intrinsically multi-user systems, they must therefore address the performance demands of their users as a group, as well as on the individual level.

Each level is explored more thoroughly below. It is important always to remember that the aim of a distributed system is to perform computation at the constituent nodes; communication between these nodes is merely a means towards this end. With current technology, the performance of such a system is almost always constrained by the time cost of the communication. In general, the time taken to transport a message between nodes far exceeds the time taken in computation producing or consuming that message.

Demands for Performance at the Individual Level

Individuals can be considered to interact with a distributed system in discrete sessions, each session is directed towards the completion of a single associated task[82]. The completion of a task involves the following activities:

- Execution of some thread of computation on the user's (client) node of the system.
- Execution of some remote operations issued from the user's node to other (server) nodes of the system.

These activities are related as follows:

- A given remote operation cannot be issued until the thread of computation has advanced to the point that the input parameters to the remote operation are available.
- There exist certain points in the execution of the thread of computation beyond which execution cannot advance until certain results of remote operations are received by the user's node.

Within this model the performance demand at the individual level is for the earliest possible time of normal termination of the user's node's thread of execution. This corresponds to the earliest possible completion of the user's assigned task.

It is assumed that the execution time taken by the computation at the user's node is independent of the communication mechanism in use. Given that this is the case, the variation in the wall-clock time for the computation will be determined solely by the delays in execution of the computation. These delays occur as a result of the following:

- The thread of computation requires a remote operation result that has not yet been received.
- Inefficient operator behaviour, including both tardiness in performing operations and wasted operations due to operator error.

[82] This model can be generalised to permit human users to interleave work on several tasks by considering each such human user as a number of users equal to the number of tasks in which he or she is engaged.

- Temporary lack of resources necessary for computation to proceed, such as expiration of the time quantum and consequent loss of the processor for a client process running on a time-sharing operating system.

This analysis ignores the last of these factors as it is not generally a property of the communication mechanism. In contrast, the first type of delay clearly is dependent on the communication mechanism.

Inefficient user behaviour may also derive from delays in operations of the communication mechanism. Human factors research generally has shown that when an operator is subject to delays of significant length while using a program, that operators attention tends to wander. In turn, this loss of focus leads to reduced efficiency of operators. Therefore, delays from other sources (such as communication operations), contribute to and exacerbate delays in this category.

If one considers a system subject only to the first type of delay, it is clear that the best performance is obtained simply by minimising the total aggregate delay. With regard to the second type of delay an additional aim is to reduce the frequency of delays of sufficient length to trigger the effect described. This aim might well be in conflict with the aim of minimising aggregate delay.

Demands for Performance at the Collective Level

Maximising the performance provided to the group of users, as a whole, is not necessarily the same as maximising performance for every user as an individual. For example, compare two distributed systems:

- A system which provides a certain level of performance, P, to each of 10 simultaneous users, but degrades to providing 10% of that performance to each of 100 simultaneous users, and
- A system which provides 50% of P to up to 100 simultaneous users.

In the case of a group of users generating demand for 100 simultaneous tasks, the second system will provide 400% better productivity.

In order to provide high performance at the collective level the percentage of time in which nodes in the distributed system are not engaged in computation must be minimised. This involves two factors:

- Maximisation of the number of processes in the system which are able to proceed at any given time.
- Maximisation of the number of such processes which are actually being executed.

Optimising the first factor involves minimising the sources of delay. This is precisely the same goal as was necessary for maximising performance at the individual level. Optimising the second factor involves three issues:

- Provision of hardware resources sufficient to cope with the load.
- Allocation and scheduling of processes to optimise use of the available hardware.
- Allocation of network resources to optimise use of the available hardware.

Consideration of the first of these is primarily in economic terms: one buys appropriate hardware subject to price and performance constraints. The second and third issues are important areas of distributed systems research, however, they are outside the scope of this thesis.

In summary, all of the factors separating performance at the collective level are outside the scope of this thesis. Therefore, this analysis neglects further discussion of collective level performance and concentrates on the individual level. Consequently, the performance impact at the collective level may be extrapolated from that at the individual level.

5.3.1 Modelling Interactions Performance in Distributed Systems

The approach to performance modelling taken in this framework is to consider client tasks as being composed of a number of subtasks, each consisting of one or more (possibly related) messages and some associated computation. As described in Section 5.1.1 of this chapter, each such sub-task is termed an interaction. Upon examining typical client programs and programming tasks one may accumulate a representative set of concrete interactions.

The performance of a remote communication mechanism may be modelled for each of the representative interactions. This modelling can account for the performance determining factors identified in the general performance model, in particular the relationships between operations within a given interaction can be accurately accounted for. From this modelling, general performance results may be extrapolated with far greater confidence than if the extrapolation were done from the basic performance quantities discussed previously.

5.3.1.1 Interaction Critical Path

Conceptually, the *critical path* of an interaction is the path from the initiation event to the termination event that takes the *longest* time to traverse. The performance of a communication mechanism supporting an interaction is determined by the time taken to traverse this critical path.

If the initiation event and the termination event are executed by different processes (which therefore have distinct clocks) then it is not possible even to measure the aggregate duration of this critical path, let alone determine the constituent edges. However, in the special case where the initiation and termination event are executed by the same process, it is possible to do both these things. Fortunately, many useful interactions (including all client/server interactions) have this property (the client executes both events) and this part of the thesis will therefore be restricted to modelling only such interactions.

The aggregate time taken to traverse the critical path of an interaction consists of the time taken to execute the computational events in that path and the time taken by message sends (including marshalling and unmarshalling *etc.*) as required to support information transfers in that path. In many cases, the contribution of computational events to the critical path traversal time is independent of the communication mechanism. The exceptions to this are cases where the communication mechanism affects the degree to which the execution of computation events can be overlapped with communication. The approach taken by this framework is to analyse the effect of such overlapping separately (in Section 5.3.2), and for the current section of the performance modelling framework to ignore overlapping.

Given these assumptions, the contribution of computational events to the interaction critical path duration is the same for all communication mechanisms. This implies that it is the time spent in message sends which determines relative performance of different communication mechanisms.

In worldwide distributed systems, transmission latencies are the primary determinant of performance. Consider a worldwide distributed system exhibiting the following characteristics:

- Average message transmission distance of 15, 000 kilometres.
- Bandwidth of one megabyte per second. This is realistic for modern private corporate networks.
- Nodes delivering a processing capacity of 100 million instructions per second.

The theoretical minimum latency determined by the speed of light for a single byte message transmitted across 15,000 kilometres, is 50 milliseconds. In practice, routing and buffering delays lead to a latency of 100 milliseconds or more. In comparison, transmission of a kilobyte message over the same distance on a typical ATM channel will take 100 milliseconds to transmit the first bit and an additional millisecond to transmit the rest of the message. In the scenario under consideration, a communication mechanism could afford 100,000 extra instructions for each of marshalling and unmarshalling of the kilobyte message, and still add only two milliseconds to the total message transmission. In this situation, a one-byte message takes 100 milliseconds and a one-kilobyte message takes 103 milliseconds.

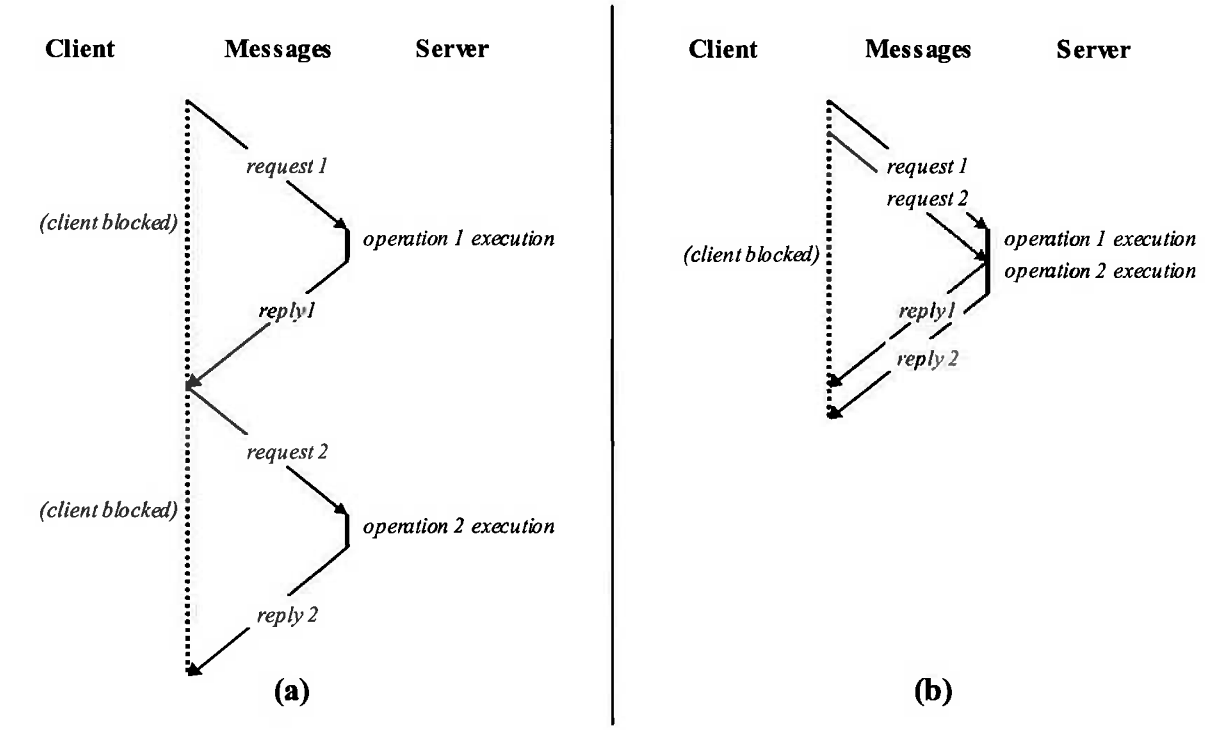

Figure 44 – Different Realisations of the Same Interaction.

Consequently, one can conclude that on the world wide scale, the dominant cost in the transmission of typically sized remote operation messages (as opposed to bulk-data transport) is the latency of getting the first bit through. This is independent of message size. Therefore, it is appropriate to model the performance in traversing an interaction's critical path in terms of the number of messages in that critical path, and to ignore the sizes of those messages.

The strategy adopted in this part of the framework is simply to count the number of messages required in the critical path of a representative set of interactions, ignoring all other factors.

5.3.1.2 Determinants of Performance

The factors determining the critical path in an interaction implemented by a given communication mechanisms are as follows:

- The messages required in support of the operations in the interaction. This corresponds to the edges on the interaction graph and is independent of the communication mechanism.
- The number of messages required in the interaction critical path provided by the communication mechanism. This is a function of the following parameters:
  - The number of operations.
  - The dependencies between operations.
  - The mapping from operations to the servers on which they are to execute.

  The function determines the effectiveness of the communication mechanism in supporting the interaction described by its parameters.

The number of messages in the critical path determines the overall latency. Different communication mechanisms can produce different critical paths, and hence different latencies, for the same interaction.

As an example, consider an interaction consisting of two operations. The ways in which two different communication mechanisms realise this interaction are shown in Figure 44. In the first case, the critical path consists of all four messages plus the execution of the two remote operations. In the second case, however, the critical path only includes *request 1* and *reply 2* plus the execution of the two remote operations. Consequently, the second mechanism has a reduced critical path message count our.

As a precursor to obtaining a representative set of interactions, the types of dependencies that may exist between operations need to be categorised.

5.3.1.2.1 Dependencies Between Operations

This model identifies five levels of dependency between pairs of operations issued from a single client, as follows:

Independent – there is no dependency between the two operations.

Order Dependent – the latter operation must execute after the former. This is not to say that both operations cannot be in progress concurrently. Rather, it is a constraint only on the order of the computational part of the remote operations.

Result dependent – one or more of the results of the former operation is used as one or more of the arguments of the latter.

Functionally dependent – one more of the arguments to the latter operation is a (non-identity) function of one or more of the results of the former operation. This includes the case where the execution of the latter operation is conditional on a function (including an identity function) of one or more of the results of the former.

Fully dependent – some action must occur on the client that uses one or more of the results of the former operation before the latter operation can commence. This in includes the case where execution of the latter operation is conditional on the result of some action that must occur on the client and which uses one more results of the former operation.

A given operation pair is categorised according to the highest level dependency between the two operations.

In many situations in which dependence is analyse, for example in parallelising or vectorising compilers, the issue of anti-dependence must also be considered. An anti-dependency occurs when the second operation modifies a value that is an input to the first operation. In this case, it must be ensured that the first operation reads the input prior to its modification by the second. Anti-dependence is not an issue in the context of remote operations. This is because the call-by-value semantics of remote operations ensure that a prior operation has a private copy of each parameter value. Consequently, both calls proceed correctly even if a later operation returns a result that is assigned to a variable passed as one of the parameters of the prior operation.

5.3.1.3 Sequential Interactions between a Client and Single Server

This subsection builds a performance model for sequential, single client, single server (*SSS*) interactions with the following constraints:

SSS-1. Interactions consist of a sequence of one or more remote operations, with dependencies between operations.

SSS-2. All operations in the interaction are issued by the same client and are served by the same server.

SSS-3. There is a total order on the execution of (the computational part of) the remote operations in the interaction. This is defined by the order in which they are issued from the client.

Although this is a restricted class of operations, it is the most important and useful class. Further subsections expand the coverage. Constraint ***SSS-3*** ensures that at most one remote operation is actually executing at any given time. However, this is a constraint only the computational part of the operation, several remote operations may be in progress at any given time. In this case, at most one of them is actually executing, the remainder are in a non executing part of remote operation service: either in transit[83] between client and server or queued waiting to be executed by the server. So remote operation execution is sequential, but remote operation service may be concurrent.

5.3.1.3.1 Groups of Operations

In this model, interactions consist of a sequence of one or more *F-groups* of operations. An F-group has the following properties:

- ***F-1.*** The F-groups in an interaction contain disjoint subsets of the operations in that interaction.
- ***F-2.*** Each F-group other than the first in an interaction contains at least one operation that is fully dependent on at least one operation in the previous F-group.
- ***F-3.*** The operations within an F-group are contiguous with respect to the total order of the interaction (***SSS-3***).
- ***F-4.*** The are no full dependencies between operations within the same F-group.
- ***F-5.*** The number of F-groups in an interaction is minimal, subject to the previous constraints.

An F-group may contain internal functional and result dependencies.

F-groups are composed of a sequence of one or more *R-groups*. These are groups of operations with the following properties:

- ***R-1.*** The R-groups in an F-group contain disjoint subsets of the operations in that F-group.
- ***R-2.*** Each R-group, other than the first in an F-group, contains at least one operation that is functionally dependent on at least one operation in the previous R-group.
- ***R-3.*** The operations within an R-group are contiguous with respect to the total order of the interaction.
- ***R-4.*** The are no functional dependencies between operations within the same R-group.
- ***R-5.*** The number of R-groups within an F-group is minimal, subject to the previous constraints.

An R-group may contain internal result dependencies.

R-groups are composed of a sequence of one or more *O-groups*. These groups have the following properties:

- ***O-1.*** The O-groups in an R-group contain disjoint subsets of the operations in that R-group.
- ***O-2.*** The operations within an O-group are contiguous with respect to the total order of the interaction.
- ***O-3.*** The are no result dependencies between operations within the same O-group.
- ***O-4.*** The number of O-groups within an R-group is minimal, subject to the previous constraints.

Recall that execution within the interaction is sequential. There are no independent (and thereby potentially concurrent) operations in this interaction.

The structure of the model may be summarised as follows. An interaction consists of one or more F-groups, each of which consists of one or more R-groups, each of which consists of one or more O-groups, each of which consists of one or more operations. At this stage, all operations are served by a single server.

Table 10 shows an example *SSS* interaction containing 14 operations structured into the following groups:

- One F-group.

[83] The state of being "in transit" includes undergoing marshalling or unmarshalling. This involves system-level computation performed by the communication mechanism. This may execute concurrently with other system level computation, or with user-level computation.

<table>
<tr><th>Operations</th><th>F-groups</th><th>R-groups</th><th>O-groups</th></tr>
<tr><td>1</td><td rowspan="14">1</td><td rowspan="3">1</td><td>1</td></tr>
<tr><td>2</td><td rowspan="2">2</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td><td rowspan="2">2</td><td rowspan="2">3</td></tr>
<tr><td>5</td></tr>
<tr><td>6</td><td rowspan="7">3</td><td rowspan="3">4</td></tr>
<tr><td>7</td></tr>
<tr><td>8</td></tr>
<tr><td>9</td><td rowspan="4">5</td></tr>
<tr><td>10</td></tr>
<tr><td>11</td></tr>
<tr><td>12</td></tr>
<tr><td>13</td><td rowspan="2">4</td><td>6</td></tr>
<tr><td>14</td><td>7</td></tr>
</table>

Table 10 – Group Structure of an Example SSS Interaction.

- Four R-groups.
- Seven O-groups

The group containment structure is indicated by the horizontal lines in Table 10.

The semi-formal group structure could be extended by a formal definition of groups as sets and dependencies as binary relations. This would be useful, in particular, if the model is used to analyse a complete application (involving 1000s of operations, interrelated in complex ways), rather than the relatively simple interactions presented as examples. In such a case, the semi-formal model would collapse under its own weight, whereas a formal mathematical model might survive. However, this thesis is attempting to draw reasonably general conclusions about the performance of specific communication mechanisms for applications in general, rather than detailed conclusions about the performance of a particular mechanism for a particular application. With such an aim, it is entirely appropriate to employ a representative sample of small interactions, and the semi-formal model is sufficient to cope with such cases.

5.3.1.3.2 Terminology

An *SSS* interaction in this model is denoted:

$$\mathrm{I}_{sss}(f,r,o,n)$$

This denotation has the following parameters:

f – the number of F-groups in the interaction.
r – the number of R-groups in the interaction.
o – the number of O-groups in the interaction.
n – the number of operations in the interaction.

These parameters are constrained as follows:

$$f \leq r \leq o \leq n \qquad \text{(Equation 1)}$$

5.3.1.3.3 Ideal Cases

For a particular communication mechanism M_X, the number of critical path messages required for an interaction $\mathrm{I}_{SSS}(f,r,o,n)$ is a function of these parameters, denoted as follows:

$$\mu_{SSS}^{M_x}(f,r,o,n)$$

There is a special communication mechanism, denoted *I*, for which **μ** gives the ideal case, with the theoretical minimum number of messages.

In the ideal case, at the beginning of each F-group, the client sends a single critical path message containing first operation in that F-group. Subsequent messages contain the rest of the operations in the F-group, and a description of the dependencies for each operation. These subsequent messages are not on the critical path. The server evaluates all the operations in order, using the dependency information as needed to generate the arguments to later operations in the group from the results of earlier operations in that group. The server then sends messages containing all the results in the F-group back to the client, only the last of these is on the critical path.

Consequently, the number of critical path messages required for an interaction is two for each F-group in that interaction, that is:

$$\mu_{SSS}^{I}(f,r,o,n)=2f \qquad \textit{(Equation 2)}$$

This ideal case assumes that it is possible to transport the executable code describing the functional dependencies between adjacent R-groups from the client to the server. The practical minimum in a system in which no code is migrated (a no code migration or NCM system) is:

$$\mu_{SSS}^{I_{NCM}}(f,r,o,n)=2r \qquad \textit{(Equation 3)}$$

That is, two messages for each R-group in the interaction. The server can use the same batched evaluation technique as before, but because there are no functional dependencies within an R-group, there is no requirement for code migration.

5.3.1.4 Extension to Multiple Servers

The model previously developed may be extended to include sequential, single client, multiple server (*SSM*) interactions. These interactions are constrained as follows:

SSM-1. Interactions consist of a sequence of one or more remote operations, with dependencies between operations.

SSM-2. All operations are issued by the same client.

SSM-3. There is a total order on the execution of (the computational part of) the remote operations in the interaction. This is defined by the order in which they are issued from the client.

Notice that ***SSM-1*** and ***SSM-3*** are the same as ***SSS-1*** and ***SSS-3*** respectively.

The extension retains the existing group structure and introduces some new groups. The first of these is called the *S-group*. In addition to containing one or more R-groups, each F-group contains one or more S-groups, with the properties:

S-1. The S-groups within an F-group contain disjoint subsets of the operations in that F-group.

S-2. The operations within an S-group are contiguous with respect to the total order of the interaction (***SSM-3***).

S-3. All operations in an S-group are served by the same server.

<table>
<tr><th>Operations</th><th>F-groups</th><th>R-groups</th><th>O-groups</th><th>O′-groups</th><th>R′-groups</th><th>S-groups</th></tr>
<tr><td>1</td><td rowspan="14">1</td><td rowspan="3">1</td><td>1</td><td>1</td><td rowspan="3">1</td><td rowspan="6">1</td></tr>
<tr><td>2</td><td rowspan="2">2</td><td rowspan="2">2</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td><td rowspan="2">2</td><td rowspan="2">3</td><td rowspan="2">3</td><td rowspan="2">2</td></tr>
<tr><td>5</td></tr>
<tr><td>6</td><td rowspan="7">3</td><td rowspan="5">4</td><td>4</td><td>3</td></tr>
<tr><td>7</td><td rowspan="2">5</td><td rowspan="2">4</td><td rowspan="2">2</td></tr>
<tr><td>8</td></tr>
<tr><td>9</td><td rowspan="2">6</td><td rowspan="2">5</td><td rowspan="2">3</td></tr>
<tr><td>10</td></tr>
<tr><td>11</td><td rowspan="2">5</td><td rowspan="2">7</td><td rowspan="2">6</td><td rowspan="4">4</td></tr>
<tr><td>12</td></tr>
<tr><td>13</td><td rowspan="2">4</td><td>6</td><td>8</td><td rowspan="2">7</td></tr>
<tr><td>14</td><td>7</td><td>9</td></tr>
</table>

Table 11 – Group Structure of an Example SSM Interaction.

S-4. The number of S-groups within an F-group is minimal, subject to the previous constraints.

There is no containment relationship between S-groups and R-groups or O-groups.

In addition to S-groups, there are two more group classes added by the model. These are derived from the existing group structure rather than directly from the interaction, and hence are referred to as derived groups. The first of these are called *R′-groups* these are contained within both an S-group and an R-group, subject to the following constraints:

R′-1. The R′-groups in an R-group contain disjoint subsets of the operations in that R-group.

R′-2. The R′-groups in an S-group contain disjoint subsets of the operations in that S-group.

R′-3. The operations within an R′-group are contiguous with respect to the total order of the interaction.

R′-4. The number of R′-groups within an R-group is minimal, subject to the other constraints.

R′-5. The number of R′-groups within an S-group is minimal, subject to the previous constraints.

Similarly, there is another derived group called an *O′-group*, this is contained within both a S-group and an O-group, and is subject to the following constraints:

O′-1. The O′-groups in an O-group contain disjoint subsets of the operations in that O-group.

O′-2. The O′-groups in an S-group contain disjoint subsets of the operations in that S-group.

O′-3. The operations within an O′-group are contiguous with respect to the total order of the interaction.

O′-4. The number of O′-groups within an O-group is minimal, subject to the other constraints.

O′-5. The number of O′-groups within an S-group is minimal, subject to the previous constraints.

As a side effect of the containment of O-groups within R-groups, O′-groups are also contained within R′-groups. The group structure is now quite complex, Table 11 shows the groups structure of an example interaction containing 14 operations and the following groups:

- One F-group.
- Four R-groups.
- Seven O-groups.

- Four S-groups.
- Seven R′-groups.
- Nine O′-groups.

The F, R and O-group containment structure is indicated by the horizontal lines in the table. Different servers are indicated by changes in shading. Finally, the dual containment of R' and O′-groups is indicated by horizontal lines, changes in shading or both.

The definition of O-groups (given in Subsection 5.3.1.3.1), implies that these interactions still only involve sequential execution of remote operations, even though the use of multiple server makes concurrency available. A later part of this framework, (Section 5.3.1.8), explores concurrency.

5.3.1.4.1 Terminology

A sequential, single-client, multiple-server (*SSM*) interaction is denoted:

$$\mathrm{I}_{SSM}(f, r, o, n, s, r', o')$$

This denotation has the following parameters:

f – the number of F-groups in the interaction.
r – the number of R-groups in the interaction.
o – the number of O-groups in the interaction.
n – the number of operations in the interaction.
s – the number of S-groups in the interaction.
r' – the number of R′-groups in the interaction.
o' – the number of O′-groups in the interaction.

The containment structure implies these parameters are constrained as follows:

$$f \le r \le o \le n \qquad (\textit{Equation 4})$$

$$f \le s \le r' \le o' \le n \qquad (\textit{Equation 5})$$

In terms of interaction structure, this class of interaction subsumes the *SSS* class, which is a special case where $s = 1$. However, it is still useful to retain two separate classes in the performance model, since communication mechanisms may deliver optimised performance in the *SSS* case.

5.3.1.4.2 Relationships

The following relationship holds between f, r, s, and r':

$$r' - r \le s - f \qquad (\textit{Equation 6})$$

Proof will proceed in stages.

First, it is necessary to show that, for an interaction with a single F-group, the following holds between r, s, and r':

$$r' \le (r + s) - 1$$

Denote the proposition that the relationship holds for some r and s as $P(r, s)$. Now consider:

$$P(1,1)$$

This clearly holds, since in this case $r' = 1$.

Assume the relationship is true for some interaction, having some indeterminate structure, x R-groups and y S-groups. Now, split one of the R-groups in this interaction, increasing x by one. If the split is made at an S-group boundary then r' remains unchanged. For this case $P(x, y) \Rightarrow P(x+1, y)$, since:

$$r' \leq (x+y)-1 \Rightarrow r' \leq (x+1+y)-1$$

If the split is made other than at an S-group boundary, then r' is increased by one because an existing R′-group is also split. For this case also, $P(x,y) \Rightarrow P(x+1,y)$, since:

$$r' \leq (x+y)-1 \Rightarrow r'+1 \leq (x+1+y)-1$$

The two cases exhaust the possibilities. Therefore, for all cases:

$$P(x,y) \Rightarrow P(x+1,y) \qquad (*)$$

S-groups and R-groups are independent, therefore, by a similar argument:

$$P(x,y) \Rightarrow P(x,y+1) \qquad (**)$$

Consequently, by induction on r and s using (*) and (**) from $P(1,1)$:

$$\forall r \geq 1, s \geq 1 : r' \leq (s+r)-1 \qquad (\mathit{where}\ f{=}1)$$

Now, the R-groups and S-groups (and R′-groups) within an F-group are independent of those in any other F-group, consequently:

$$\forall r \geq 1, s \geq 1, f \geq 1 : r' \leq (s+r)-f$$

Eliding the quantifiers and rearrangement gives the result:

$$r' - r \leq s - f$$

A similar argument is used to show the relationship between the number of O-groups, O′-groups, F-groups and S-groups, namely:

$$o' - o \leq s - f \qquad (\mathit{Equation\ 7})$$

5.3.1.4.3 Ideal Cases

For a particular communication mechanism M_X, the number of critical path messages required for an interaction $\mathrm{I}_{SSM}(f,r,o,n,s,r',o')$ is a function of the interaction parameters, denoted as follows:

$$\mu_{SSM}^{M_X}(f,r,o,n,s,r',o')$$

There is a special communication mechanism, denoted *I*, for which μ gives the ideal case, with the theoretical minimum number of messages.

In the ideal case, at the beginning of each F-group, the client sends a single critical path message to the server for the first S-group in that F-group. This message contains the first operation in the F-group. Subsequent messages to the server contain the rest of the operations in that F-group, and a description of the dependencies for each operation. These subsequent messages are not on the critical path. The server then evaluates all the operations in the S-group in order, using the dependency information as needed to generate the arguments to later operations in the group from the results of earlier operations in that group. Where there is more than one S-group in the F-group, the server sends a critical path message to the server for the next F-group. This contains the first operation in that S-group and any results from the current S-group on which the operation depends. Subsequent messages are sent to that server containing the rest of the operations in that F-group, a description of the dependencies for each operation, and all the results computed in the current S-group. This step might be termed *inter-server request chaining*. As was previously the case, the subsequent messages are not on the critical path. The server receiving this message then proceeds to execute the operations in its S-group in the same way as the first server. Finally, the server for the last S-group sends messages containing all the results in the F-group back to the client, only the last of these would be on the critical path. Consequently, the number of critical path messages required for an interaction in the ideal case is the sum of two terms, as follows:

- Two messages for each F-group in the interaction
- A message for each additional S-group within an F-group (over and above the first S-group in each F-group, which is required by the containment criteria). This is the difference between the number of S-group and the number of F-groups, $s - f$, which must be non-negative, since $f \leq s$.

Therefore:

$$\mu^{I}_{SSM}(f,r,o,n,s,r',o') = 2f + (s-f) \\ = s + f \qquad \textit{(Equation 8)}$$

As previously, this ideal case assumes that it is possible to migrate the executable code describing the functional dependencies between adjacent R-groups from the client to the servers.

The practical minimum number of critical path messages in a linguistically heterogeneous system is deduced by the following reasoning:

- Assuming all operations were executed by the same server,. The number of messages required is $2r$ (as given previously).
- However, the interaction spans multiple servers and it may be that some of the R-groups straddle across more than one S-group. At each such straddle point an additional message is required, from one server to the next.
- The number of these straddle points is the difference between the number of R′-groups and the number of R-groups, that is $r' - r$.

Therefore:

$$\mu^{I_{NCM}}_{SSM}(f,r,o,n,s,r',o') = 2r + (r'-r) \\ = r + r' \qquad \textit{(Equation 9)}$$

5.3.1.5 Modelling Administrator Delegation Interactions

An administrator delegation (*AD*) interaction, described in Section 5.1.3.1.3 above, involves a single request, submitted by a client, delegated through *a* administrators, and delegated by the last administrator to be served by a server. The result of this server is then transmitted (directly or indirectly) back to the client. This interaction is denoted:

$$\mathrm{I}_{AD}(a)$$

Where the parameter, *a*, gives the number of administrators in the interaction.

5.3.1.5.1 Ideal Case

The minimum number of messages to support this interaction is the sum of the following terms:

- One message from the client to the first administrator.
- One message for each delegation, that is, *a* messages.
- One final message from the server to the client.

Therefore:

$$\mu^{I}_{AD}(a) = a + 2 \qquad \text{(Equation 10)}$$

No code migration is required here, so the ideal case with no code migration is the same:

$$\mu^{I_{NCM}}_{AD}(a) = \mu^{I}_{AD}(a) = a + 2 \qquad \text{(Equation 11)}$$

5.3.1.5.2 Equivalence to General Sequential Multiple Server Interaction

The administrator delegation interaction is actually a sub-class of the general sequential multiple server interaction (Section 5.3.1.4). The delegation interaction can be considered to be a sequence of $a + 1$ operations, each executed on a different server, with each operation result dependent (and therefore order-dependent) on its predecessor, and with no functional or full dependencies. Therefore:

$$f = 1, \quad r = 1, \quad o = a + 1, \quad n = a + 1, \quad s = a + 1, \quad r' = a + 1 \quad o' = a + 1$$

Substituting these values into the ideal message count for the sequential multiple server interaction gives:

$$\begin{aligned}\mu^{I}_{SSM}(f, r, o, n, s, r', o') &= s + f \\ &= a + 1 + 1 \\ &= a + 2 \\ &= \mu^{I}_{AD}(a)\end{aligned}$$

Similarly, substitution into the ideal case with no code migration gives what one would expect:

$$\begin{aligned}\mu^{I_{NCM}}_{SSM}(f, r, o, n, s, r', o') &= r + r' \\ &= 1 + a + 1 \\ &= a + 2 \\ &= \mu^{I_{NCM}}_{AD}(a)\end{aligned}$$

However, even in the presence of this equivalence it is appropriate to model the delegation interaction separately. The reason is the existence of communication mechanisms that support request delegation but do not support more general inter-server request chaining. Therefore, a given mechanism might attain the ideal in the delegation case but not in the more general case.

5.3.1.6 Layered Servers

As described previously, layered server architectures are becoming increasingly more important in distributed systems. The simplest layered architecture has three participants:

- a client,
- an application server, and,
- a resource server.

The interaction proceeds as follows:

1. The client sends a request to the application server.
2. The application server performs some computation, then sends a request to the resource server.

3. The resource server receives this request, performs some computation, and then sends a reply back to the application server.
4. The application server receives the reply, performs some further computation, and then sends reply back to the client.

Four messages are required for this interaction.

This example interaction can be viewed as the composition of an interaction between the client and application server and another interaction between the application server and the resource server (with the former in the role of client). Both of these are *SSS* interactions parameterised as follows:

$$\mathrm{I}_{SSS}(1,1,1,1)$$

Now, the message count for the overall interaction (4 messages) can be obtained by simple addition of the messages counts for each of the composed interactions (2 messages in each case).

The ability to add message counts in this way is surprisingly general. It holds with interactions involving an arbitrary number of server layers where the client and higher layer servers are involved in fully general *SSM* interactions with their respective interaction partners. The constraints under which it holds are:

- The interactions to be composed must be sequential in nature, that is, *SSM* (including *SSS*) or *AD*,
- The interactions to be composed must be "proper" interactions in the sense developed previously in this framework (Section 5.1.1), in particular, they must achieve coordinated termination internally.

Relaxation of the first constraint is discussed later, in Section 5.3.1.8.6. The second constraint is a limitation of the model: it is possible for an overall sequential interaction that achieves coordinated termination to be composed of *sub-interactions* that do not. Nevertheless, such cases are rare, and probably represent a high risk programming approach.

In consequence, the performance characteristics of a communication mechanism supporting most layered server interactions are a function the performance characteristics in support of *SSM* interactions. Therefore, the layered server characteristics do not require separate modelling.

5.3.1.7 One-way Requests

One-way requests in *SSS* and *SSM* interactions can be modelled by adding an extra parameter, *m*, giving the number of one-way request operations in the interaction. Therefore, the *SSS* interaction is denoted:

$$\mathrm{I}_{SSS}(f,r,o,n,m)$$

Similarly, the *SSM* interaction is denoted:

$$\mathrm{I}_{SSM}(f,r,o,n,m,s,r',o')$$

Note that the total number of operations (both one-way and ordinary) in the interaction is still *n*, so there are now *n–m* ordinary operations.

5.3.1.7.1 Ideal Cases

One way requests are a restricted case of ordinary requests, and the ideal cases for both *SSS* and *SSM* are independent of the number of requests, so the ideals remain the same, that is:

$$\mu^{I}_{SSS}(f,r,o,n,m)=2f \qquad \textit{(Equation 12)}$$

$$\mu_{SSS}^{I_{NCM}}(f,r,o,n,m)=2r$$ (*Equation 13*)

$$\mu_{SSM}^{I}(f,r,o,n,m,s,r',o')=s+f$$ (*Equation 14*)

$$\mu_{SSM}^{I_{NCM}}(f,r,o,n,m,s,r',o')=r+r'$$ (*Equation 15*)

5.3.1.8 Concurrency from Independent Operations

A limited form of concurrency can be introduced by the removal of the constraint requiring each operation to be order dependent on its predecessor. This leads to a new class of concurrent operation, single client, multiple server (*COSM*) interactions with the following properties:

COSM-1. Interactions consist of a sequence of one or more remote operations, with dependencies between operations.

COSM-2. All operations are issued by the same client.

COSM-3. There is a total order defined by the order in which operations are issued from the client

This is the *SSM* interaction without the total ordering constraint on operation execution.

The interaction retains the existing *SSM* group structure with the addition of a further kind of basic group, the *I-group*. One or more I-groups are contained within each O-group, these are sequences of operations subject to he following constraints:

I-1. The I-groups in an O-group contain disjoint subsets of the operations in that O-group.

I-2. All operations in an I-group are order dependent on all operations in the previous I-group. This replaces the constraint in *SSM* interactions requiring total order dependence.

I-3. The operations within an I-group are contiguous with respect to the order of operation issue (***COSM-3***).

I-4. The are no order dependencies between operations within the same I-group.

I-5. The number of I-groups within an O-group is minimal, subject to the previous constraints.

I-groups are therefore groups of independent operations, which may be executed concurrently.

This model also adds two derived groups, *I′-groups* and *IR′-groups*. I′-groups are contained within both an I-group and an S-group, subject to the following constraints:

I′-1. The I′-groups in an I-group contain disjoint subsets of the operations in that I-group.

I′-2. The I′-groups in an S-group contain disjoint subsets of the operations in that S-group.

I′-3. The operations within an I′-group are contiguous with respect to the order of operation issue.

I′-4. The number of R′-groups within an R-group is minimal, subject to the other constraints.

I′-5. The number of R′-groups within an S-group is minimal, subject to the previous constraints.

The operations in an I′-group are hence independent operations executed by the same server, not necessarily in the order of issue.

IR′-groups are derived from both I-groups and R′-groups; by application of the following constraints:

IR′-1. A contiguous group of one or more I-groups wholly contained in an R′-group is an IR′-group.

IR′-2. Each I-group wholly or partially containing more than one R′-group is an IR′-group.

<table>
<tr><th>Operations</th><th>F-groups</th><th>R-groups</th><th>O-groups</th><th>I-groups</th><th>IR′-groups</th><th>I′-groups</th><th>O′-groups</th><th>R′-groups</th><th>S-groups</th></tr>
<tr><td>1</td><td rowspan="14">1</td><td rowspan="3">1</td><td>1</td><td>1</td><td rowspan="3">1</td><td>1</td><td>1</td><td rowspan="3">1</td><td rowspan="6">1</td></tr>
<tr><td>2</td><td rowspan="2">2</td><td>2</td><td>2</td><td rowspan="2">2</td></tr>
<tr><td>3</td><td>3</td><td>3</td></tr>
<tr><td>4</td><td rowspan="2">2</td><td rowspan="2">3</td><td rowspan="2">4</td><td rowspan="2">2</td><td rowspan="2">4</td><td rowspan="2">3</td><td rowspan="2">2</td></tr>
<tr><td>5</td></tr>
<tr><td>6</td><td rowspan="7">3</td><td rowspan="5">4</td><td>5</td><td>3</td><td>5</td><td>4</td><td>3</td></tr>
<tr><td>7</td><td rowspan="4">6</td><td rowspan="4">4</td><td rowspan="2">6</td><td rowspan="2">5</td><td rowspan="2">4</td><td rowspan="2">2</td></tr>
<tr><td>8</td></tr>
<tr><td>9</td><td rowspan="2">7</td><td rowspan="2">6</td><td rowspan="2">5</td><td rowspan="2">3</td></tr>
<tr><td>10</td></tr>
<tr><td>11</td><td rowspan="2">5</td><td>7</td><td rowspan="2">5</td><td>8</td><td rowspan="2">7</td><td rowspan="2">6</td><td rowspan="4">4</td></tr>
<tr><td>12</td><td>8</td><td>9</td></tr>
<tr><td>13</td><td rowspan="2">4</td><td>6</td><td>9</td><td rowspan="2">6</td><td>10</td><td>8</td><td rowspan="2">7</td></tr>
<tr><td>14</td><td>7</td><td>10</td><td>11</td><td>9</td></tr>
</table>

Table 12 – Group Structure of an Example COSM Interaction.

IR′-3. The number of IR′-groups within an R-group is minimal, subject to the previous constraints.

The defining characteristic of an IR′-group is that it does not contain any dependencies that span multiple servers. IR′-groups are contained in R-groups and contain I-groups (and therefore, contain I′-groups).

An example interaction of this form is shown in Table 12. This shows the group structure as previously. The interaction contains the following groups:

- One F-group.
- Four R-groups.
- Seven O-groups.
- Ten I-groups.
- Six IR′-groups.
- Eleven I′-groups
- Nine O′-groups
- Seven R′-groups
- Four S-groups

In this example, IR′-groups 1,2,3,5 and 6 are formed by application of the ***IR′-1*** constraint, and group 4 is formed by application of the ***IR′-2*** constraint.

5.3.1.8.1 Terminology

A client, multiple-server interaction with operation level concurrency (*COSM*) is denoted:

$$\mathrm{I}_{COSM}(f,r,o,i,n,m,s,r',o',i',ir')$$

This denotation has the following parameters:

f – the number of F-groups in the interaction.
r – the number of R-groups in the interaction.
o – the number of O-groups in the interaction.

i – the number of I-groups in the interaction.
n – the number of operations in the interaction (including one-way operations).
m – the number of one-way operations in the interaction.
s – the number of S-groups in the interaction.
r' – the number of R′-groups in the interaction.
o' – the number of O′-groups in the interaction.
i' – the number of I′-groups in the interaction.
ir' – the number of IR′-groups in the interaction.

These parameters are constrained as follows:

$$f \le r \le o \le i \le m \le n$$
$$f \le s \le r' \le o' \le i' \le n$$
$$r \le ir' \le i$$
$$r' \le (r+s)-1$$
$$o' \le (o+s)-1$$

5.3.1.8.2 Relationships

The following relationship holds in *COSM* interactions:

$$ir' \le r' \qquad \textit{(Equation 16)}$$

This relationship between IR′-groups and R′-groups is demonstrated as follows:

- For IR′-groups subject to constraint ***IR′-1***, there is at most one IR′-group per R′-group
- An IR′-group subject to constraint ***IR′-2***, contains (wholly or partially) at least two R′-groups, an R′-group subject to this constraint is contained (wholly or partially) in at most two IR′-groups. Therefore, there are at most two IR′-groups for every pair of R′-groups.
- IR′-groups subject to constraints ***IR′-1*** and ***IR′-2*** form disjoint sets.

Therefore, one concludes that the number of IR′-groups is less than or equal to the number of R′-groups, in spite of the fact that R′-groups are not contained in IR′-groups. It is also possible to prove this relationship by an inductive process similar to that used in Section 5.3.1.4.2.

5.3.1.8.3 Ideal Cases

The ideal for this interaction is deduced as follows:

- Assuming all operations are executed by the same server, the number of messages required is $2f$ (as given previously).
- However, the interaction spans multiple servers. An additional message is required between each adjacent pair of IR′-groups within an R-group to enforce the ordering of that pair.
- The number of these pairs is the difference between the number of IR′-groups and the number of R-groups, that is, $ir' - r$.

Therefore:

$$\mu^{I}_{COSM}(f,r,o,i,n,m,s,r',o',i',ir') = 2f + (ir' - r) \qquad \textit{(Equation 17)}$$

Now, recall

$$ir' \le r' \qquad \textit{(Equation 16)}$$

$$\therefore ir' - r \le r' - r$$

And

$$r' - r \leq s - f \qquad \text{(Equation 6)}$$

Therefore:

$$ir' - r \leq s - f$$
$$\therefore 2f + (ir' - r) \leq 2f + (s - f)$$

Consequently, the ideal for *COSM* is less than or equal to that for *SSM*:

$$\mu^{I}_{COSM}(f,r,o,i,n,m,s,r',o',i',ir') \leq \mu^{I}_{SSM}(f,r,o,n,m,s,r',o')$$

The practical minimum in a system in which code is not mobile is deduced from the above result by the following reasoning:

- Assuming all operations were executed by the same server, the number of messages required is $2r$ (as given previously).
- However, the interaction spans multiple servers. An additional message is required between each adjacent pair of IR′-groups within an R-group to enforce the ordering of that pair.
- The number of these pairs is the difference between the number of IR′-groups and the number of R-groups, that is *ir'-r*.

Therefore:

$$\mu^{I_{NCM}}_{COSM}(f,r,o,i,n,m,s,r',o',i',ir') = 2r + (ir' - r) \qquad \text{(Equation 18)}$$
$$= r + ir'$$

Recall :

$$ir' \leq r' \qquad \text{(Equation 16)}$$

$$\therefore r + ir' \leq r + r'$$

That is, the ideal for *COSM* with no code migration is less than or equal to that for *SSM*:

$$\mu^{I_{NCM}}_{COSM}(f,r,o,i,n,m,s,r',o',i',ir') \leq \mu^{I_{NCM}}_{SSM}(f,r,o,n,m,s,r',o')$$

That is, the introduction of concurrency can improve the ideal in the heterogeneous case as well.

5.3.1.8.4 Application to Single Server Interactions

The *COSM* interaction is applicable even in the case where there is only a single, single-threaded server in the interaction. The reason is that some communication mechanism may be able to service several independent requests concurrently but service order dependent requests serially. To model this characteristic, there is another interaction, termed concurrent operations, single client, single server (*COSS*). This is denoted and defined as follows:

$$\mathrm{I}_{COSS}(f,r,o,i,n,m) = \mathrm{I}_{COSM}(f,r,o,i,n,m,1,r,o,i,r)$$

The ideal cases for *COSS* are:

$$\mu^{I}_{COSS}(f,r,o,i,n,m) = 2f + (r - r)$$
$$= 2f$$
(*Equation 19*)

$$\mu^{I_{NCM}}_{COSS}(f,r,o,i,n,m) = 2r + (r - r)$$
$$= 2r$$
(*Equation 20*)

This is the same as for *SSM*, which is expected since the ideals assume that order-dependent requests (but not the operations thereof) may be in progress concurrently.

5.3.1.8.5 Multi-threaded Servers

The contract-based software engineering model prohibits multi-threaded access to objects. However, multi-threaded servers can be implemented as a collection of concurrently accessible single threaded objects. A typical structure has one dispatcher object and several worker objects. Clients send requests to the dispatcher, which acts as an administrator forwarding requests to idle workers. If the dispatcher and worker objects are located on the same computer (or even the same LAN), then communication latencies in the dispatching of operations are sufficiently small that they may be ignored in the message count. The overall effect is of a single, apparently multi-threaded, server object.

The use of multi-threaded servers permits the concurrent execution of independent operations to the same server. In cases where the execution of these operations does not engage in remote interaction, the multi-threading will not lead to a reduction in overall message count. The concurrent execution may lead to a gain in performance, but this is independent of the communication mechanism. A more interesting case occurs if the execution of some of the operations involves some nested remote interaction. This case is discussed next.

5.3.1.8.6 Application to Layered Server Interaction

Previous discussion of layered server interaction assumed that remote operation execution is sequential and consequently that any sub-interactions within those operations are part of the overall interaction sequence. The overall impact is that the critical path message counts for any sub-interactions are added to the critical path message count for the top-level interaction to give the overall critical path message count.

The introduction of concurrent execution of operations, even in the limited form above, changes all this. Suppose the operations in an I-group execute concurrently and spawn sub-interactions. In this case, the overall critical path message count for that I-group is the message count for the I-group at the top level plus the largest critical path message count for any of the sub-interactions. If there is more than one sub-interaction in the I-group, then only the longest sub-interaction is on the critical path, the remainder do not contribute to the overall critical path message count at all. This is a potentially significant saving in a large interaction.

As with the case of sequential interaction with layered servers, modelling a concurrent layered server system would not give any additional information about the properties of a communication mechanism, these properties are already modelled in the *COSM* and *COSS* case. However, concurrent interaction with layered servers demonstrates the effect of even the small amount of concurrency in the *COSM* case and the *COSS* with a multi-threaded server.

5.3.1.9 Other Forms of Concurrency

This subsection describes two general forms of concurrency that are excluded from the model.

5.3.1.9.1 Concurrency between Independent Groups

There are cases where there are adjacent independent groups within an interaction. Each group contains intra-group dependencies, but there are no inter-group dependencies. The operations in these groups could be in progress concurrently. Catering for this level of concurrency greatly increases the complexity of the model, for very little improvement in capability. Therefore, there is no modelling of this situation.

It should be noted that a situation in which there are several groups of independent operations proceeding concurrently is a good candidate for the use of multi-threading in the client. This can be supported within the contract-based approach by splitting the client into multiple objects in the same way as is done to create a multi-threaded server. The introduction of multi-threading of this kind is independent of the communication mechanism, this is another reason to consider such modelling unnecessary.

5.3.1.9.2 Concurrency between Loosely Dependent Groups

An even more complex case arises where there is loose dependency between groups. These groups can be in progress concurrently most of the time, but occasionally need to synchronise to maintain dependencies. There is no modelling of this situation for the same reasons as given for the previous case.

5.3.2 Overlapped Computation

This section builds a simple model of the degree to which a client thread is able to continue computation whilst communication initiated by that thread is in progress. Blocking is influenced by two kinds of factors, as follows:

- Factors that cause the client to block, thereby preventing it from continuing computation.
- Delays in sending messages that lead to increased duration of blocking at some later stage.

Messages may be delayed due to batching, buffering or other strategies. Each of the above kinds of factors is discussed in the subsections below.

5.3.2.1 Client Blocking Factors

These can be evaluated by identifying sets of operations for which blocking ensues. These sets are defined by an intentional constraint given by the associated blocking criterion, the sets used in this framework are as follows:

COM – set of all communication operations.

CR – subset of ***COM*** where results are of reference types.

CN – subset of ***COM*** where results are of non-reference types.

USE – set of communication operations having results that are used locally prior to the value of the result becoming available. This is a subset of ***COM***.

UR – subset of ***USE*** where results are of reference types. This is a subset of ***CR***.

UN – subset of ***USE*** where results are of non-reference types. This is a subset of ***CN***.

SND – set of communication operations having results that are sent in network messages prior to the value of the result becoming available. This is a subset of ***COM***

SR – subset of ***SND*** where results are of reference types. This is a subset of ***CR***.

SN – subset of ***SND*** where results are of non-reference types. This is a subset of ***CN***.

SP – subset of ***SND*** where results are sent only to the server responsible for producing them.

SPR – subset of ***SP*** where results are of reference types. ***SPR*** is also a subset of ***SR*** where results are sent only to the server responsible for producing them.

SPN – subset of ***SP*** where results are of non-reference types. ***SPN*** is also a subset of ***SN*** where results are sent only to the server responsible for producing them.

SC – subset of ***SND*** where results are sent only to a server other than the server responsible for producing them.

SCR – subset of ***SC*** where results are of reference types. ***SCR*** is also a subset of ***SR*** where results are sent only to a server other than the server responsible for producing them.

SCN – subset of ***SC*** where results are of non-reference types. ***SCN*** is also a subset of ***SN*** where results are sent only to a server other than the server responsible for producing them.

The support for overlapped computation provided by communication mechanisms is assessed by identifying the sets for which blocking ensues under that communication mechanism.

In theory, the assessment result for a given communication mechanism could be any of the large number of set expressions derived from the basic sets listed above. In practice, however, only a small number of such expressions are represented in extant communication mechanisms, as follows:

1 – ***COM*** – in which all communication operations cause the client to block.

2 – $\boldsymbol{CN \cup UR \cup SR}$ – in which all communication operations with non-reference results cause the client to block, as do all uses and sends of unavailable reference results.

3 – $\boldsymbol{USE \cup SND}$ – in which all uses and sends of unavailable results cause the client to block.

4 – $\boldsymbol{CN \cup UR \cup SCR}$ – in which all communication operations with non-reference results cause the client to block, as do all uses of results and sends of unavailable reference results to servers other than the server producing them.

5 – $\boldsymbol{USE \cup SC}$ – all uses of results and sends of unavailable results to servers other than the server producing them cause the client to block.

6 – $\boldsymbol{USE \cup SCN}$ – all uses of results and sends of unavailable non reference results to servers other than the server producing them cause the client to block.

7 – ***USE*** – in which only uses of unavailable results cause the client to block.

Notice that blocking factors and the critical path message count are not necessarily independent, in that blocking factors may limit the client's ability to submit requests. This in turn limits the ability for requests to be concurrently in-progress, and thus may inflate the critical path message count. However, blocking factors also model the ability for the client to overlap computation with communication, something not accounted for in the critical path message count.

It is clear that sets **2-7** are all subsets of ***COM***, which is effectively the universal set of communication operations. By inspection, set **7** is a subset of sets **5** and **6.** Furthermore:

$\boldsymbol{USE = UR \cup UN}$

$\boldsymbol{SND = SR \cup SN}$

$\therefore \boldsymbol{USE \cup SND = UR \cup UN \cup SR \cup SN}$

Now:

$\boldsymbol{UN \cup SN \subseteq CN}$ (since there may be operations with results that are never used or sent).

$\therefore \boldsymbol{UR \cup UN \cup SR \cup SN \subseteq CN \cup UR \cup SR}$ (adding $\boldsymbol{UR \cup SR}$ to both sides).

Therefore:

$\boldsymbol{USE \cup SND \subseteq CN \cup UR \cup SR}$

That is, set **3** is a subset of set **2**. Also:

$\boldsymbol{CN \cup UR \cup SCR \subseteq CN \cup UR \cup SR}$ (since $\boldsymbol{SCR \subseteq SR}$).

That is, set **4** is also a subset of set **2**. In addition:

$USE \cup SC = UR \cup UN \cup SCR \cup SCN$

Recall

$UN \cup SN \subseteq CN$

$\therefore\ UN \cup SCN \cup SPN \subseteq CN$ (since $SN = SCN \cup SPN$).

$\therefore\ UN \cup SCN \cup SPN \cup UR \cup SCR \subseteq CN \cup UR \cup SCR$

$\therefore\ UN \cup SCN \cup UR \cup SCR \subseteq CN \cup UR \cup SCR$

$\therefore\ UN \cup UR \cup SCN \cup SCR \subseteq CN \cup UR \cup SCR$

Therefore:

$USE \cup SC \subseteq CN \cup UR \cup SCR$ (since $USE = UN \cup UR$ and $SC = SCN \cup SCR$).

That is, set **5** is a subset of set **4**.

$USE \cup SC \subseteq USE \cup SND$ (since $SC \subseteq SND$).

That is, set **5** is also as subset of set **3**. Finally:

$USE \cup SCN \subseteq USE \cup SC$ (since $SCN \subseteq SC$).

That is, set **6** is a subset of set **5**.

Summarising the discussion, the following subset relationships exist between the seven sets listed above:

$COM \supseteq CN \cup UR \cup SR \supseteq USE \cup SND \supseteq USE \cup SC \supseteq USE \cup SCN \supseteq USE$

$COM \supseteq CN \cup UR \cup SR \supseteq CN \cup UR \cup SCR \supseteq USE \cup SC \supseteq USE \cup SCN \supseteq USE$

Consequently, with the exception of sets **3** and **4**, there is a total ordering on the cardinality of the sets indicated by their number. In respect of sets **3** and **4**, the expectation is that in most cases set **3** represents a more restrictive blocking regime than does set **4**. The main reason for this view is the programmer can eliminate *CN* by modifying operations to return reference type results. Consequently, the support for overlapping computation provided by a communication mechanism is assessed by classification into one of the above sets, with higher numbered sets being preferred.

5.3.2.2 Message Transmission Delays

Modelling the performance of an interaction by means of analysis if the critical path is accurate as long as messages on the critical path are transmitted immediately the corresponding remote operations requests and replies are issued by program elements. However, some communication mechanisms might not transmit all request and replies immediately. For example, several requests to a given server might be batched into a single message to reduce consumption of network resources. This will delay commencement of processing (by the server) of all the requests, and therefore delay the consequent transmission of results back to the client, thereby increasing the overall critical path. The overall effect is a diminution of the opportunity for overlapped execution of client and server as well as an increase in the duration of the critical path.

Delaying message transmission is significant for both request and reply messages, though probably less so for the latter. There are some differences between the request and reply cases, which are therefore assessed independently

5.3.2.2.1 Request Transmission Policy

The assessment categories for a middleware mechanism's policy on request transmission delay are as follows:

1 – *Lazy* – a request is transmitted only when the client that submitted that request blocks waiting for a reply to a request that has not been transmitted. This approach minimises the

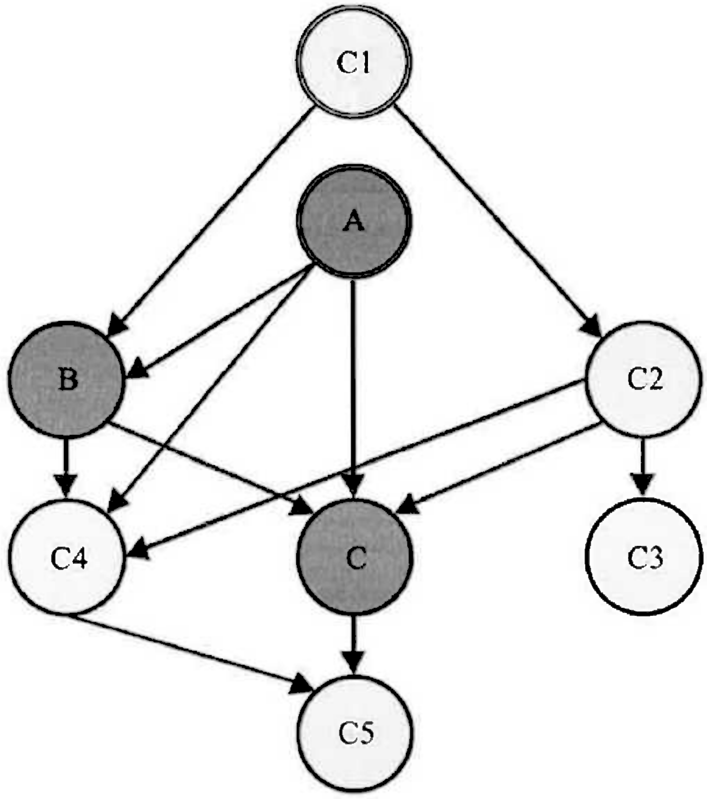

Figure 45 – Dependencies in an Example Interaction.

requirement for network resources, but reduces the opportunity for overlapping the execution of client and server.

2 – *Balanced* – the system attempts to strike a balance between laziness and strictness, attempting to conserve network resources whilst not reducing the opportunity to overlap execution of client and server.

3 – *Eager* – a request is transmitted immediately it is submitted.

5.3.2.2.2 Reply Transmission Policy

Suppose a server has received and is processing a group of requests from a given client. The server might opt to batch all the replies. The assessment categories for a middleware mechanism's policy on reply transmission delay are as follows:

1 – *Lazy* – a groups of replies destined for a given client are sent from the server only if there are no requests for that client in progress or pending on the server. This approach minimises the requirement for network resources, but may delay the commencement of client processing of all but the last of the results transmitted in the replies.

2 – *Balanced* – the system attempts to strike a balance between laziness and strictness, attempting to conserve network resources whilst not overly delaying transmission of replies.

3 – *Eager* – a reply is transmitted immediately it is submitted.

5.3.3 A Detailed Example Applying the Performance Model

The performance modelling has examined four kinds of factors determining the performance of distributed systems interactions, as follows:

- Critical path message count.
- Client blocking factors.
- Request transmission delays.
- Reply transmission delays.

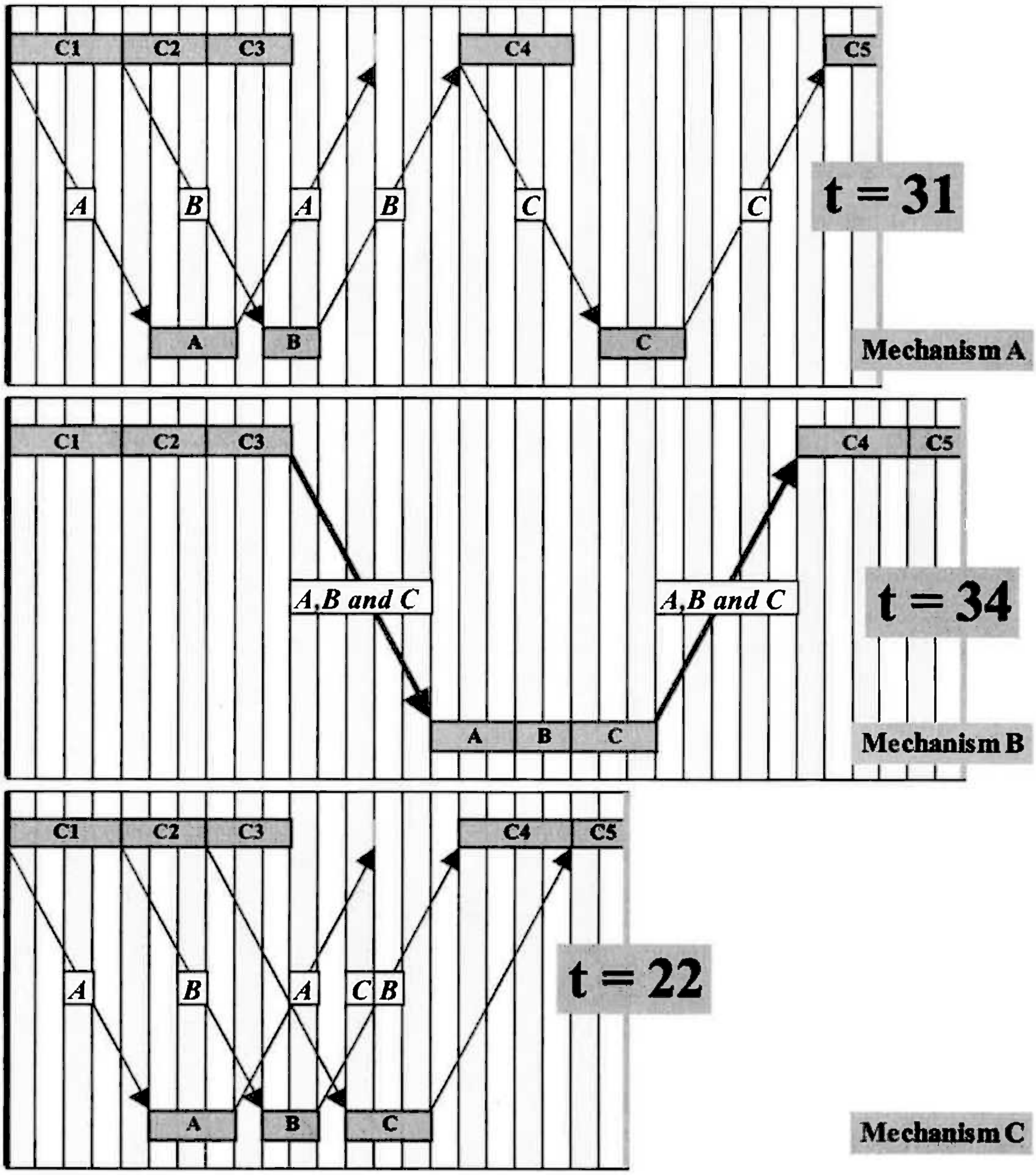

Figure 46 – Comparative Timing Diagrams with a Message Latency of 5.

These factors will affect the overall performance of different interactions to different degree. To explore this, a relatively simple example is analysed in detail.

The example involves a single client and server. Three remote computations, *A*, *B* and *C*, are executed by the server. In addition, five local computations *C1*, *C2*, *C3*, *C4* and *C5* are executed by the client. The dependencies amongst the local and remote computations are shown in Figure 45.

The effects of the various performance factors are illustrated by the timing diagrams in Figure 46. These illustrate the history of the interaction for three different communication mechanisms. These communication mechanisms are as follows:

Mechanism A – in which messages are sent eagerly but the client blocks when it attempts to send an unavailable result as a parameter.

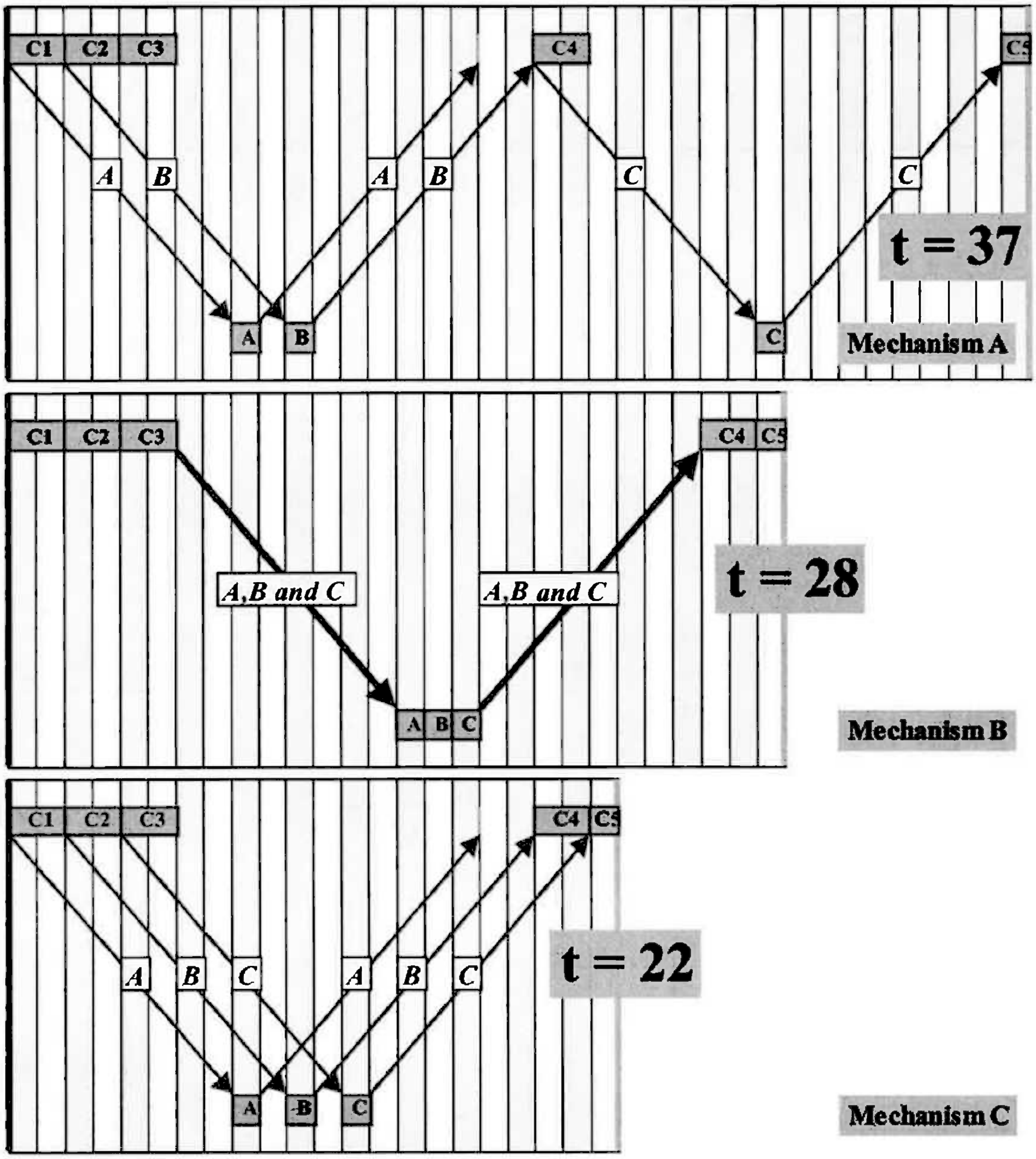

Figure 47 – Comparative Timing Diagrams with a Message Latency of 8.

Mechanism B – in which messages are sent lazily but the client can, without blocking, send the result of an incomplete request as a parameter to a subsequent request. This mechanism therefore allows result interdependent requests to be scheduled concurrently, whereas mechanism A only allows only order-interdependent requests to be scheduled concurrently.

Mechanism C – which combines eager message transmission with the ability to send unavailable result as parameters without blocking.

The latency of message transmission and the various computation times are constant across all three mechanisms in Figure 46.

Inspection of the first two diagrams in Figure 46 indicates for mechanism A, the critical path message count is four (requests B and C are in the critical path), whereas for mechanism B, it is two. However, mechanism A outperforms mechanism B in spite of having a larger number of messages in the critical path. This occurs because the ratio between message transmission latency and computation time is quite low. The third diagram indicates that the combination of both optimisation techniques leads to the best performance of the three mechanisms, as would be expected to be the case.

The ratio between the computation times and the message transmission latency shown in Figure 46 is typical only of very low latency networks, and is certainly not appropriate for wide area networks or even for large local networks.

Figure 47 contains timing diagrams for the same interaction but in which the ratio between message transmission and computation times has been increased by approximately a factor of five. Even this ratio is unrealistic for wide area systems, but the trend is clearly that with higher latencies, the relative importance of eager message transmission is decreased and the relative performance of reducing the critical path message count is increased. As before, the combination of the two optimisations outperforms either in isolation.

5.3.4 Priority of Performance Determinants

In wide area networks where message transmission latencies are high, that minimising the number of message in the critical path is by far the most importance optimisation strategy. Consequently, this factor is accorded the highest priority in evaluating the potential performance of worldwide communication mechanisms.

Minimisation of request transmission delays is considered the second most significant factor, since a delay in request transmission delays both the server computation that is triggered by that request and any computation using the result of the request. In most cases, results are not produced gratuitously, so it is likely that there is a computation awaiting the result.

Minimisation of reply transmission delays is considered the next most significant factor, since such delays also delay the start of computations that require the result contained in the reply.

Finally, the impact of client blocking on the ability of the client to overlap computation with communication is the least significant factor (given that one excludes the impact of client blocking on the critical path message count). The ability of clients to perform significant useful computation without performing strict operations on result is generally limited. There are cases in which the client performs an extensive local computation that is independent of the results of requests it has submitted. However, such cases represent client activity that is in large part independent of any communication in which the client is engaged. As such, these computation are best executed as separate client threads.

5.4 Assessment Checklists

This section summarises the framework with a series of checklists are used in the assessment of communication mechanisms. These are applied in the assessment of extant communication mechanisms given in the next chapter. The checklists are given in tabular form, Table 13 summarises the application section of the framework, Table 14 the software engineering section and Table 15 the performance section.

Capability	**Assessment**	
Replies	Boolean	Indicating presence of reply messages distinct from requests.
Early Reply	Boolean	Indicating whether early reply is supported.
Reply Matching	Levels	1 – No support, 2 – Tagging, 3 – Full support.
Reply Synchronisation	Levels	1 – Asynchronous, 2 – Explicit Synchronous, 3 – Implicit, 3 – Hybrid.
Reply Kinds	Set	Includes: local reference, remote reference, non-reference.
Coordinated Termination	Levels	1 – No support, 2 – Supported, 3 – Automatic.
Syntactic Disruption	Levels	1 – Significant, 2 – Limited, 3 – None.
Semantic Disruption	Levels	1 – Significant, 2 – Consistent, with exceptions, 3 – Consistent.
One-way Requests	Boolean	Indicates presence of support.
Request Forwarding	Levels	1 – No support, 2 – Support via reflection, 3 – Linguistic, 4 – Automatic.
Peer Symmetry	Boolean	Indicating whether peer symmetry is supported.
Reply Entailment	Levels	1 – No support, 2 – Simple client/server support, 3 – Extended peer reply support.
Reply Races	Levels	1 – No support, 2 – Winner value only, 3 – All reply values.
Multiple Replies	Boolean	Indicates whether a given request message can have more than one corresponding reply message.
Interaction Structure	Levels	1 – Fixed, 2 – Regular, 3 – Recursive, 4 – General.
Linguistic Heterogeneity	Boolean	Indicating whether linguistic heterogeneity can be supported.

Table 13 – Application Assessment Checklist.

Capability	Assessment	
Type Checking Timing	Levels	1 – Dynamic, 2 – Hybrid, 3 – Static (dynamic only as strictly necessary).
Type Checking Clustering	Levels	1 – Incompatible with clustering, 2 – Compatible with clustering.
Assertion Coverage	Levels	1 – None, 2 – Propositional, 3 –Predicate, 4 – Predicate multi-object.
Intrinsic Assertion Specification	Boolean	Indicating whether specification is intrinsic.
Inseparable Assertion Specification	Levels	1 – Separable, 2- Partially separable, 3 – Inseparable.
Declarative Assertion Specification	Levels	1 – No support, 2 – Operational via functions, 3 – Declarative propositional terms + functions, 4 – Declarative predicate terms.
Early Assertion Checking	Levels	1 – Wholly dynamic, 2 – Partially static.
Precondition Synchronisation	Levels	1 – No synchronisation semantics, 2 – Reflection, 3 – Server queueing 4 – Client wait, 5 – Client wait and server side queuing.
Asynchronous Contract Completion	Levels	1 – No support, 2- Exception on failure, 3 – Notification on completion, 4 – Optional synchronisation.
Reply Specification Coverage	Levels	1 – No support, 2 – Simple (client/server), 3 – Extended.
Multiple Reply Entailment	Boolean	Indicates whether a contract can specify more than one reply for a given message.
Interaction Structure Specification	Levels	1 – Fixed, 2 – Regular, 3 – Recursive, 4 – General.
Intrinsic Reply Specification	Boolean	Indicates whether specification is intrinsic, as preferred.
Inseparable Reply Specification	Boolean	Indicates whether specification is inseparable, as preferred
Declarative Reply Specification	Boolean	Indicates whether specification is declarative, as preferred.
Positive Reply Specification	Boolean	Indicates whether specification is positive, as preferred.
Reply Sending Enforcement	Boolean	Indicates whether system enforces obligations to send replies.
Illegal Reply Handling	Levels	1 – No enforcement, 2 – Filtering, 3 – Rejection, 4 – Prevention
Illegal Reply Rejection Timing	Levels	1 – Dynamic, 2 – Static.
Extra Reply Handling	Levels	1 – No enforcement, 2 – Filtering, 3 – Rejection, 4 – Prevention
Illegal Reply Rejection Timing	Levels	1 – Dynamic, 2 – Static.
Asynchronous Method Call Exception Handling	Boolean	Indicates whether handlers can be specified for exceptions raised in asynchronous remote method calls.
Exception Handling Contexts	List of Levels	1 – Object, 2 – Reply Based, 3 – Caller Frame, 4 – Caller.
Exception Handling Flexibility	Boolean	Indicates whether the programmer can choose from various alternative for exception handling, allowing reduced context to be traded off for more relaxed synchronisation.

Table 14 – Software Engineering Assessment Checklist.

Interaction		Ideal Cases		Assessment
Description	***Denotation***	**Ideal**	**No Code Migration**	
Sequential, Single Client, Single Server	$I_{SSS}(f,r,o,n,m)$	$2f$	$2r$	Function giving message count on critical path.
Sequential, Single Client, Multiple Server	$I_{SSM}(f,r,o,n,m,s,r',o')$	$s+f$	$r+r'$	
Administrator Delegation	$I_{AD}(a)$	$a+2$	$a+2$	
Concurrent Operations, Single Client, Single Server	$I_{COSS}(f,r,o,i,n,m)$	$2f$	$2r$	
Concurrent Operations, Single Client, Multiple Server	$I_{COSM}(f,r,o,i,n,m,s,r',o',i',ir')$	$2f+(ir'-r)$	$r+ir'$	

Capability	**Assessment**	
Overlapped Computation	Levels	Levels 1-7 as described above (in Section 5.3.2 on page 161).
Request Transmission Policy	Levels	1 – Lazy, 2 – Balanced, 3 – Eager.
Reply Transmission Policy	Levels	1 – Lazy, 2 – Balanced, 3 – Eager.

Table 15 – Performance Assessment Checklist.

Chapter Six

Survey of Communication Mechanisms

This chapter applies the framework developed in the previous chapter to the assessment of current communication mechanisms in distributed object systems. This is in contrast to a survey such as [Philippsen 1995], which concentrates on concurrent object-oriented programming languages, exclusive of "programming environments that are oriented towards distributed programming". Rather than being exhaustive, the survey is intended to be representative. It includes both commercially oriented middleware systems and research systems. The survey is organised into the following parts:

- Section 6.1 discusses mechanisms derived from synchronous remote procedure call (RPC).
- Section 6.2 discusses mechanisms in which the synchronisation exhibited in the RPC mechanisms can be deferred.
- Section 6.3 discusses one way message-passing systems, which eliminate the concept of reply and mechanisms based upon one way message-passing in which the responsibility to synchronise with the arrival of remote replies is able to be delegated to an entity other than the caller of the method generating the reply.
- Section 6.4 discusses mechanisms providing reflective facilities for extension .
- Section 6.5 discusses mechanisms based on migration of program entities.

Notice that several categories are distinguished by the way in which replies are handled. This illustrates the high significance of reply scheduling facilities in middleware systems.

6.1 Synchronous RPC Derivatives

Remote procedure call (RPC) [Nelson 1981, Birell and Nelson 1984], was the original high-level middleware paradigm. RPC aims to provide a remote operation call mechanism that closely resembles the local procedure call paradigm. In particular, the execution of an RPC is synchronous with respect to the caller. The main difference between an RPC and a local procedure call is that parameters and results of an RPC are transmitted by copying rather than by reference. This is because the caller and target for an RPC are typically in different address spaces. In the RPC paradigm, the targets of remote calls are collections of functions and procedures. These collections are often called remote interfaces. There is generally no way to obtain a reference to a remote interface which may then be passed to other parts of the system, therefore, remote interfaces are not objects (nor are they classes). Consequently, RPC is unsuitable as a mechanism for distributed object systems. However, it is possible to derive mechanisms suitable for distributed object system from RPC. These derivatives are assessed below. The assessment is summarised in Table 16, using the checklists given at the end of the previous chapter. Where assessment is by classification into levels, the assessed, *a* and maximum, *m*, level values are given in the form (*a*/*m*).

Capability	**Java RMI**	**CORBA RMI**	**+Forwarding**	**+One-way RMI**	**+Early Reply**	**Parallel RMI**
Replies	True					
Early Reply	False				True	False
Reply Matching	Full support (3/3)					
Reply Synchronisation	Implicit (3/3)					
Reply Kinds	All	Remote & non-reference	(follows base mechanism)			All
Coord. Termination	Automatic (3/3)			Supported (2/3)		
Syntactic Disruption	None (3/3)					Significant (1/3)
Semantic Disruption	Consistent (3/3)					
One-way Requests	False			True	False	
Request Forwarding	False		True	False		
Peer Symmetry	False					
Reply Entailment	Simple (2/3)					
Reply Races	No support (1/3)					
Multiple Replies	False					
Interaction Structure	Fixed (1/4)					
Linguistic Hetero.	True					

Capability	**Early Reply**	**RMI, Request Forwarding**	**One-way RMI**
Type Checking Timing	Static (3/3)		
Type Checking Clustering	True		
Assertion Coverage	Not Applicable		
Intrinsic Assertion Specification			
Inseparable Assertion Specification			
Declarative Assertion Specification			
Early Assertion Checking			
Precondition Synchronisation			
Asynchronous Contract Completion			
Reply Specification Coverage	Simple (2/3)		
Multiple Reply Entailment	False		
Interaction Structure Specification	Fixed (1/4)		
Intrinsic Reply Specification	True		
Inseparable Reply Specification			
Declarative Reply Specification			
Positive Reply Specification			
Reply Sending Enforcement			
Illegal Reply Handling	Prevention (4/4)		
Illegal Reply Rejection Timing	Not Applicable		
Extra Reply Handling	Rejection (3/4)	Prevention (4/4)	
Extra Reply Rejection Timing	Dynamic (1/2)	Not Applicable	
Asynchronous Method Call Exception Handling	Not Applicable		
Exception Handling Contexts			
Exception Handling Flexibility			

Interaction	**RMI, Early Reply**	**Multi-threaded RMI**	**Parallel RMI**	**One-way RMI**	**Request Forwarding**
$\mathrm{I}_{SSS}(f,r,o,n,m)$	$2(n+m)$		$2o$	$2n$	$2(n+m)$
$\mathrm{I}_{SSM}(f,r,o,n,m,s,r',o')$			$2o'$		
$\mathrm{I}_{AD}(a)$	$2a+2$				$a+2$
$\mathrm{I}_{COSS}(f,r,o,i,n,m)$	$2(n+m)$	$2I$	$2o$	$2n$	$2(n+m)$
$\mathrm{I}_{COSM}(f,r,o,i,n,m,s,r',o',i',ir')$			$\min(2i, 2o')$		
Overlapped Client Computation	(1/7)	(7/7)	(1/7)		
Request Transmission Policy	Eager (3/3)				
Reply Transmission Policy					

Table 16 –Assessment Checklists– Synchronous RPC Derivatives.

6.1.1 Remote Method Invocation

Remote method invocation (RMI) is the direct application of RPC into the object arena. The targets of remote method invocations are remote objects rather than collections of remote objects. References to

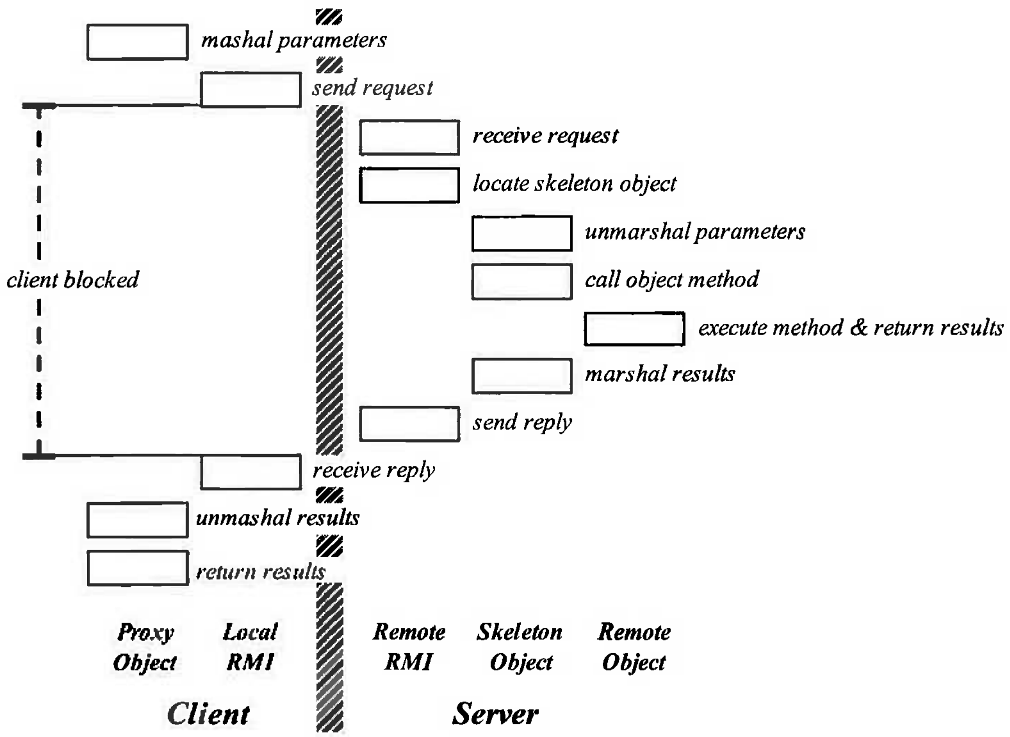

Figure 48 – Remote Method Invocation.

these remote objects may be passed around in much the same way as references to local objects. Furthermore, references to objects exporting remote object interfaces can be passed, by reference, in remote method invocations. Passing of other objects retains the copy-based semantics of RPC.

Almost all distributed object systems include an RMI communication mechanism. In fact, in many cases, RMI may be the only communication mechanism. Furthermore, something similar to the proxy/skeleton implementation approach is generally used, even in systems adopting other communication mechanisms. The most well known examples of RMI systems are CORBA [OMG 1997] and Java RMI [Sun 1997a]. Two other well known mechanisms, DCE [OSF 1992] and DCOM [Brown and Kindel 1996] provide facilities similar to RMI but without full object semantics for call targets. The object-oriented programming language Ada 95 [ISO 1995], supports an RMI-like facility in a standard extension (annexure E).

RMI systems provide the illusion of local method calls to objects that are, in fact, located in a separate address space, possibly on another computer. In order to call methods of a remote (server) object, the client must first obtain a local object representing that remote object. In many systems, these local objects are called *proxies*. Other, equivalent terms include *stub* and *surrogate*. Proxies for remote objects of a given server class are objects of a class derived from the interface of that class. In particular, the proxy has methods corresponding exactly to each of the methods in the server interface. To effect a call of a given method of a remote object, the client simply calls the corresponding method of that object's (local) proxy. Any actual parameters to be passed to the remote invocation are passed to the proxy method call, which sends them to the remote object over the network. Any results returned by the remote operation are sent back to the client and then returned as results of the proxy method call. From the point of view of the client, the appearance is of a synchronous local method call, returning any results contained in the reply message as results of the call.

The implementation of the call is hidden within the proxy method. The steps taken in executing a remote method call are shown in Figure 48. The proxy method marshals the parameters, then supplies a request message to be sent over the network by the local RMI core. This message is received by the remote RMI core, which locates the *skeleton* object associated with the remote object. There is a skeleton object corresponding to each remotely accessible object on a server. Each skeleton object declares a method, typically named *dispatch*, which takes a request message as a parameter, the remote (server) RMI core calls this method, passing the request message. The dispatch method of the skeleton determines the method to be called, unmarshals the parameters in the request, and then calls the appropriate method on the remote object. This method call executes and then returns zero or more result to the call of *dispatch*. These results are marshalled into a reply message and passed back to the client via the RMI core, the network and the proxy, as shown in the figure. From the point of view of the server, the appearance is of a call to a particular method in which the request values are provided via parameters and the reply values are supplied via the return mechanism. Finally, in an object-oriented language, the sub-typing mechanism can be used to integrate proxies seamlessly within the client program. Consequently, syntactic disruption is eliminated (with the possible exception of object creation, where a suitable proxy must be substituted for a local object). Semantic disruption is also minimised.

RMI supports request/reply structured communication. Replies are sent via the return construct of the server-side implementation language. On the client side, reply matching is seamlessly supported through integration with the method return mechanism. Reply synchronisation is implicit because remote method calls are synchronous. There is no support for one-way requests or request forwarding. Peer symmetry is not supported, peer interactions must be encoded in client/server interactions. Reply entailment is supported at the simple client/server level. RMI does not require code- migration, consequently, it is wholly compatible with linguistic heterogeneity.

RMI supports static type checking, and is compatible with clustering of dynamic type checks. None of the common RMI systems (Java RMI, CORBA etc) support assertions, though there is no fundamental incompatibility with such support. RMI supports reply contract specification at the simple level. These specifications are given in IDL interfaces to remotely callable objects, and are thereby intrinsic to, and inseparable from, those objects. Furthermore, they are both declarative and positive in form in all common RMI systems. Both reply sending and reply validity are automatically enforced by the communication mechanism. Furthermore, because of the structure of communication, reply validity enforcement reaches the maximum assessed level. Finally, because communication is synchronous, asynchronous exceptions are not a possibility and a standard synchronous exception handling facility is sufficient.

RMI requires a two-message round-trip between the client and the server for each remote operation. This includes operations that are conceptually one-way requests. The critical path message count is therefore $2(n + m)$ for the *SSS*, *SSM*, *COSS* and *COSM* interactions. In respect of administrator delegation, the reply must return to the client via all the administrators, the critical path message count is hence $2a + 2$. These message counts are worse than for most of the alternatives – performance is the Achilles Heel of RMI in worldwide systems.

6.1.2 One-way RMI

RMI follows RPC in being a two-way communication. In this communication, the caller (client) sends a request containing parameter values and the target (server) sends a reply containing result values. The caller blocks after sending the request message until the reply arrives. In certain situations, there may be no need for the reply message, a one-way communication from caller to target may be sufficient. Furthermore, because there is no reply message, there is no need for the sender to block. One way RMI supports this communication pattern.

Many RMI systems support one-way RMI in addition to standard RMI. Two prominent examples are CORBA, in which methods can be marked by the reserved word ***oneway*** and DCE, in which remote operations can be marked by the reserved word ***maybe***. Ada 95, annexure E [ISO 1995], has a one way remote operations mechanism called asynchronous RPC.

The addition of one-way operations to RMI leads to the possibility of exceptions being raised asynchronously in remote operation execution. Furthermore, in the majority of one-way RMI mechanisms (such as CORBA **oneway** methods), there is no means to capture and handle these exceptions. One-way operations improve performance in comparison to RMI by itself, the critical path message counts for *SSS*, *SSM*, *COSS* and *COSM* are $2n$ rather than $2(n + m)$. The message count for administrator delegation remains the same, at $2a + 2$.

6.1.3 Multi-threaded RMI

One drawback with synchronous RMI is that the client (caller) is blocked whilst the RMI is in progress. One approach to overcoming this situation is for the client to create a child thread to perform each RMI asynchronously. The parent thread can then proceed with further activity, including starting additional RMI child threads, whilst the child blocks. The parent synchronises with the relevant child when it needs the result(s) of that child's RMI.

Two prominent examples of systems oriented towards multi-threaded RMI are Java, which has language support for threads, and DCE, which has a threads sub-system.

As discussed previously, the unnecessary use of multiple threads is problematic in software engineering terms. The problems are both general (as discussed in Section 2.3.1.5.6) and specific to the application of the contract based approach to software engineering (Section 4.1.3). Therefore, the framework for assessment, in Section 5.2.2, specifically disqualifies mechanisms that involve using extra threads for the sole purpose of achieving asynchronous RMI (as opposed to any other use of extra threads). Consequently, the multi-threaded RMI approach is not assessed with respect to software engineering. Furthermore, this approach is the same as single threaded RMI is respect of application issues, so the only assessment is of performance.

With regard to performance, the critical path message counts are the same as for RMI in the case of the *SSS* and *SSM* interactions, because of the ordering constraints between adjacent operations. For the *COSS* and *COSM* interactions, adjacent operations within an I-group are independent, and may therefore proceed concurrently. Consequently, the critical path message count for these interactions is $2I$, where I gives the number of I-groups. The performance for administrator delegation is the same as for RMI. Finally, multi-threaded RMI permits the highest level of overlapped computation because client execution proceeds in a separate thread to communication. Consequently, it is blocked only when a remote result is required, there are no structural constraint imposed by the communication mechanism on overlapped computation.

6.1.4 Delegated Call

The concurrent object-oriented programming language Hybrid [Neierstrasz 1987], supports a limited from of multi-threaded RPC termed delegated call. In this approach, objects are contained within groups called domains. There may be several threads of execution within each domain, however at most one of these may be executing at any given time. Hence, the threads within a given domain are coroutines [Marlin 1980]. The currently executing thread in a domain may initiate a delegated call to the method of an object in another domain. This call blocks the caller thread, but in contrast to standard RMI (which is also available in Hybrid), execution may then switch to another coroutine in the domain.

Recall that one problem with general multi-threading is that it precludes the ability to reason about program behaviour based on an invariant describing the stable state of an object. This occurs because there might be two or more threads executing a given object's methods, and only the first to start can expect to find the object stable. In Hybrid, the (possibly several) activities within a domain execute as coroutines, rather than as concurrent threads. Context switches between these coroutines take place at delegated calls.

Coroutines are a restriction of the general multi-threading case, but unfortunately, it does not address the problem. In a review of the Hybrid project [Papathomas and Konstantas 1990], the following is stated:

> *Consider an object A which calls a method of another object B which is in the same domain. If B issues a delegated call, it may trigger the execution of a method of another object in its domain. In particular, the delegated call may cause the execution of another method of A. This interleaved execution of A's methods, which occurs at points not controlled by A but by objects that are called by it, may cause the execution of A's method to fail. There is no guarantee that a class invariant, which is a necessary precondition for the correct execution f A's methods holds when A calls B or any other object in its domain that performs a delegated call.*

That is, that the problem occurs with coroutine in combination with delegated call much as it does with general multi-threading. Consequently, the delegated call mechanism is not considered separately from multi-threaded RMI.

6.1.5 RMI with Request Forwarding

Request forwarding enables a server to delegate responsibility for sending a reply to a given RMI to another server. An example of the use of request forwarding by intermediate servers acting as administrators is given in the previous chapter (Sections 5.1.3.1.3 and 5.1.4.6.3).

The Hybrid [Neierstrasz 1987] programming language includes a language construct for request forwarding. The ABCL [Yonezawa *et.al.* 1987] and Mentat [Grimshaw 1993] object-based programming languages also include request forwarding support at the language level. It is also possible to achieve request forwarding by reflective manipulation of the data structures of the communication mechanism, as in Eiffel // [Caromel 1993].

The addition of request forwarding to the basic RMI mechanism improves the performance for the administrator delegation (*AD*) interaction, and has no other effect. For this interaction request forwarding achieves the optimum result of $a + 2$.

6.1.6 Parallel RMI

In a standard (single-threaded) RMI system, a client is at any time in one of two states, as follows:

- *Executing* – engaged in a local operation.
- *Blocked* – awaiting the reply for a single remote operation.

Parallel RMI extends the *blocked* state so that the client may have more than one remote operation in progress. There is still the strict alternation between the two states: an executing client becomes blocked by sending one or more remote operations request (potentially to different servers), a blocked client resume executing when all requested remote operations have completed. No dependencies are permitted between the operations submitted in a group.

ABCL [Yonezawa *et.al.* 1987, Shibayama, Ichisugi and Yonezawa 1990], supports parallel RMI with a dedicated construct, but the construct has not been adopted widely. One major problem is that it entails major disruption of existing programs, both syntactically and semantically. This may explain the lack of widespread adoption.

Parallel RMI affects only performance, it does not impact application or software engineering issues. It allows a batch of remote operation requests to be issued in a single communication operation, after which the client is blocked until all operations in the batch have completed. This mechanism differs from RMI only in terms of performance and only for the *SSS*, *SSM*, *COSS* and *COSM* interactions, *AD* interaction performance and support for overlapped execution is unchanged. There can be no result or higher dependencies between operations in the batch. Operations within the batch issued to a given server execute in the order of issue. Operations to different servers within a batch execute concurrently.

For the *SSS* interaction, a group of operations containing internal order dependencies but no internal result or higher dependencies (an O-group) may be executed as a single batch. Consequently, the critical path message count is $2o$, where o gives the number of O-groups. The only difference between *SSS* and *COSS* is the potential concurrency between operations in an I-group. With parallel RMI in both *COSS* and *SSS* interactions, all operations in a O-group are in progress concurrently (albeit they execute sequentially in *SSS*). Now, since I-groups are contained within O-groups, any concurrency within I-groups does not further reduce the message count. Consequently, the message count for the *COSS* interaction is the same as for *SSS*.

For the *SSM* interaction, the situation is somewhat different, because of the need to enforce ordering constraints between operations on different servers. Consequently, only contiguous partitions of O-groups that execute on the same server can be batched. These partitions are O′-groups in the performance model, the critical path message count is $2o'$, where o' gives the number of O′-groups.

For the *COSM* interaction there are two choices for batching of operations, either each I-group can be executed as a single batch, or each O′-group can be executed as a batch. The choice of which is minimal depends on whether there are less I-groups or less O′-groups in the interaction, which varies between interactions, since there is no containment relationship between these two kinds of groups. Consequently, the critical path message count for this interaction is given by MIN($2i$, $2o'$).

6.1.7 RMI with Early Reply

It is possible to extend RMI with an early reply operations. This extension is purely a server side facility, no difference is perceived by clients. The introduction of the early reply facility has a slight affect on the software engineering assessment in comparison to that for RMI.

6.2 Deferred Synchronous Mechanisms

In systems derived from RPC/RMI, clients block on starting a remote communication operation. It may be possible to continue execution in another thread, but the communication operation itself is synchronous.

An alternative is for the communication mechanism to support the deferral of blocking on a given remote operation until the result of that operation is required by the client. This is called the *deferred synchronous* approach. The assessment of the mechanisms in this section is summarised in Table 17.

6.2.1 Futures

Futures [Halstead 1985], were introduced as a means of expressing concurrent computation in *Multilisp*, a multi-processor programming system. This system allowed function calls to execute concurrently with their caller. The caller synchronises with each call only when the result computed by that call is needed, rather than at the point of the call, as would be the case in a sequential system.

Futures are used in the distributed communication context to express caller/callee concurrency, for example in the *Cronus* [Walker, Floyd and Neves 1990] system. Remote invocations in futures based system such as *Cronus* have two phases:

1. *Request submission* – the remote operation to invoke and the arguments and target for that operation are specified. A (request) message containing that information is sent over the network to the target. The client may continue executing once the message has been provided to the operating system for transmission. The client is provided with a variable of type ***future*** [*x*], where *x* is the type of the result of the call. The variable provided to the client is termed a future, which it may later use in claiming the result of the call.

<table>
<tr><th>Capability</th><th>Futures</th><th>Wait-by-Necessity & SCOOP</th><th>Batched Futures</th><th>Batched Futures + Promises</th><th>CORBA DII</th><th>Mentat & Legion</th></tr>
<tr><td>Replies</td><td colspan="6">True</td></tr>
<tr><td>Early Reply</td><td colspan="5">False</td><td>True</td></tr>
<tr><td>Reply Matching</td><td colspan="6">Full support (3/3)</td></tr>
<tr><td>Reply Sync.</td><td>Expl.. (2/3)</td><td colspan="2">Implicit (3/3</td><td>Hybrid (3/3)</td><td colspan="2">Explicit (2/3)</td></tr>
<tr><td>Reply Kinds</td><td>All</td><td>All Reference</td><td>Remote</td><td colspan="2">Remote & Non-reference</td><td>All</td></tr>
<tr><td>Coord. Termination</td><td colspan="6">Supported (2/3)</td></tr>
<tr><td>Syntactic Disruption</td><td>Signif. (1/3)</td><td colspan="2">None (3/3)</td><td>Limited (2/3)</td><td>Signif. (1/3)</td><td>None (3/3)</td></tr>
<tr><td>Semantic Disruption</td><td>Cons. (3/3)</td><td colspan="3">Consistent, with exceptions (2/3)</td><td colspan="2">Consistent (3/3)</td></tr>
<tr><td>One-way Requests</td><td colspan="6">True</td></tr>
<tr><td>Request Forwarding</td><td colspan="5">No support (1/4)</td><td>Auto (4/4)</td></tr>
<tr><td>Peer Symmetry</td><td colspan="6">False</td></tr>
<tr><td>Reply Entailment</td><td colspan="6">Simple (2/3)</td></tr>
<tr><td>Reply Races</td><td>All vals (3/3)</td><td colspan="5">No support (1/3)</td></tr>
<tr><td>Multiple Replies</td><td colspan="6">False</td></tr>
<tr><td>Interaction Structure</td><td colspan="6">Fixed (1/4)</td></tr>
<tr><td>Linguistic Hetero.</td><td colspan="6">True</td></tr>
</table>

<table>
<tr><th>Capability</th><th>CORBA DII</th><th>Wait by Necessity</th><th>SCOOP</th><th>Futures, Batched Futures, Mentat & Legion</th></tr>
<tr><td>Type Checking Timing</td><td>Dynam. (1/3)</td><td colspan="3">Static (3/3)</td></tr>
<tr><td>Type Checking Clustering</td><td>False</td><td colspan="3">True</td></tr>
<tr><td>Assertion Coverage</td><td rowspan="9">Not Applicable</td><td colspan="2">Predicate (3/4)</td><td rowspan="9">Not Applicable</td></tr>
<tr><td>Intrinsic Assertion Specification</td><td colspan="2">True</td></tr>
<tr><td>Inseparable Assertion Specification</td><td colspan="2">Inseparable (3/4)</td></tr>
<tr><td>Declarative Assertion Specification</td><td colspan="2">Propositional terms + functions (3/4)</td></tr>
<tr><td>Early Assertion Checking</td><td colspan="2">Wholly dynamic (1/2)</td></tr>
<tr><td>Precondition Synchronisation</td><td>Reflective (2/5)</td><td>Client wait (4/5)</td></tr>
<tr><td>Asynchronous Contract Completion</td><td colspan="2">No support (1/4)</td></tr>
<tr><td>Reply Specification Coverage</td><td colspan="4">Simple (2/3)</td></tr>
<tr><td>Multiple Reply Entailment</td><td colspan="4">False</td></tr>
<tr><td>Interaction Structure Specification</td><td colspan="4">Fixed (1/4)</td></tr>
<tr><td>Intrinsic Reply Specification</td><td colspan="4" rowspan="5">True</td></tr>
<tr><td>Inseparable Reply Specification</td></tr>
<tr><td>Declarative Reply Specification</td></tr>
<tr><td>Positive Reply Specification</td></tr>
<tr><td>Reply Sending Enforcement</td></tr>
<tr><td>Illegal Reply Handling</td><td colspan="4">Prevention (4/4)</td></tr>
<tr><td>Illegal Reply Rejection Timing</td><td colspan="4">Not Applicable</td></tr>
<tr><td>Extra Reply Handling</td><td colspan="4">Prevention (4/4)</td></tr>
<tr><td>Extra Reply Rejection Timing</td><td colspan="4">Not Applicable</td></tr>
<tr><td>Async., Call Exception Handling</td><td>True</td><td colspan="2">False</td><td>True</td></tr>
<tr><td>Exception Handling Contexts</td><td>Reply (2/4), Caller (4/4)</td><td colspan="2" rowspan="2">Not Applicable</td><td>Reply (2/4), Caller (4/4)</td></tr>
<tr><td>Exception Handling Flexibility</td><td>True</td><td>True</td></tr>
</table>

<table>
<tr><th>Interaction</th><th>Futures & CORBA DII</th><th>Wait by Necessity & SCOOP</th><th>Batched Futures</th><th>Batched Futures with Promises</th><th>Mentat & Legion</th></tr>
<tr><td>$\mathrm{I}_{SSS}(f,r,o,n,m)$</td><td>$2o$</td><td>$\geq 2o$</td><td>$\geq 2r$</td><td colspan="2">$2r$</td></tr>
<tr><td>$\mathrm{I}_{SSM}(f,r,o,n,m,s,r',o')$</td><td>$2o'$</td><td>$\geq 2o'$</td><td>$\geq 2r'$</td><td>$2r'$</td><td>$r+r'$</td></tr>
<tr><td>$\mathrm{I}_{AD}(a)$</td><td colspan="4">$2a+2$</td><td>$a+2$</td></tr>
<tr><td>$\mathrm{I}_{COSS}(f,r,o,i,n,m)$</td><td>$2o$</td><td>$\geq 2o$</td><td>$\geq 2r$</td><td colspan="2">$2r$</td></tr>
<tr><td>$\mathrm{I}_{COSM}(f,r,o,i,n,m,s,r',o',i',ir')$</td><td>min(2$i$,2$o'$)</td><td>$\geq 2o'$</td><td>$\geq 2r'$</td><td>$2r'$</td><td>$r+ir'$</td></tr>
</table>

<table>
<tr><td>Overlapped Computation</td><td>(3/7)</td><td>(2/7)</td><td>(4/7)</td><td>(5/7)</td><td>(7/7)</td></tr>
<tr><td>Request Transmission Policy</td><td colspan="2" rowspan="2">Eager (3/3)</td><td rowspan="2">Lazy (1/3)</td><td colspan="2" rowspan="2">Eager (3/3)</td></tr>
<tr><td>Reply Transmission Policy</td></tr>
</table>

Table 17 – Assessment Checklist– Deferred Synchronous Mechanisms.

2. *Result claim* – the client executes a claim operation on the future corresponding to a previous call. This retrieves the result of that call, blocking until that result is available.

In these systems, the synchronisation associated with retrieving the result of a remote invocation is deferred until the second phase. Continuation of execution after the first phase for a given call may include the first phase for subsequent calls, hence the client may have several calls in progress with this mechanism. A remote operation invocation using futures is shown in Figure 49.

The *Cronus* system, which introduced the application of futures to distributed systems, is not an object based system, although an equivalent object based system could be built. ABCL [Yonezawa *et.al.* 1987], is an object based system incorporating futures. Promises [Liskov and Shrira 1988], are a futures like facility within the overall context of the Mercury system [Liskov *et.al.* 1988]. Recent development has brought asynchronous calls and futures to Java [Caromel and Vayssière 1998, Caromel, Klauser and Vayssière 1998, Kerry Falkner, Coddington and Oudshoorn 1999, Goldberg *et. al.* 1999]

Futures are purely a client-side mechanism, operations invoked by a client using futures appear to the server as standard RMI calls. Reply sending operations are thus the same as for RMI, an early reply operation could be supported but for this survey, use of the standard return operation is assumed. Reply matching is fully supported by the mechanism, in that the retrieval of the reply associated with a request is effected by an operation on the object (a future) denoting that request. Reply synchronisation occurs at the point of call to the claim operation, and therefore is explicit. Coordinated termination is fully supported, but is not automatic, since it is possible to leave futures unclaimed. One way requests are supported, in effect, because there is no need to ever claim the future associated with a remote operation. Request forwarding, in being a server-side operation, is orthogonal to a futures mechanism. As for early reply, this survey assumes there is no request forwarding support, though there is no fundamental incompatibility. Futures are oriented towards client/server communication, there is no support for peer symmetry. Finally, reply entailment reaches the simple multi-reply level, because it is possible to issue several claims on a given future.

Future are similar to parallel RMI in one sense, in that both mechanisms allow multiple remote method calls to be in progress concurrently. Futures are less disruptive to existing code, in there is no need to group a collection of remote calls together to have then processed concurrently, as is the case with parallel RMI. There is still significant syntactic and semantic disruption however, because claims of results must be introduced into the code and modification made to each remote method call.

In respect of software engineering issues, futures follows RMI mechanisms as described previously, with the exception of support for handling exceptions raised by asynchronous remote operations. Such exceptions are raised (synchronously) at the point where the future denoting a given remote call is claimed. Futures are first class objects in the client implementation language, they can therefore be passed around. The caller of a given remote operation could claim the future for that operation, or it could store that future in a variable or return it, for some other method activation to claim. Futures therefore reach the highest level of support for exception handling of asynchronous remote calls.

In respect of performance for the test case extended interactions, futures are equivalent to parallel RMI. With parallel RMI, a collection of remote operations is treated as single communication operation, during which the client is suspended. With futures, each remote operation is a communication operation, and the client is suspended only when a claim is made. This means that, with futures, it is possible for there to be client computation overlapped with communication.

6.2.2 CORBA Dynamic Invocation Interface

CORBA does not support deferred synchronous communication at the language (stubs) level,. However, at a slightly lower level, the CORBA dynamic invocation interfaces, supports formulation

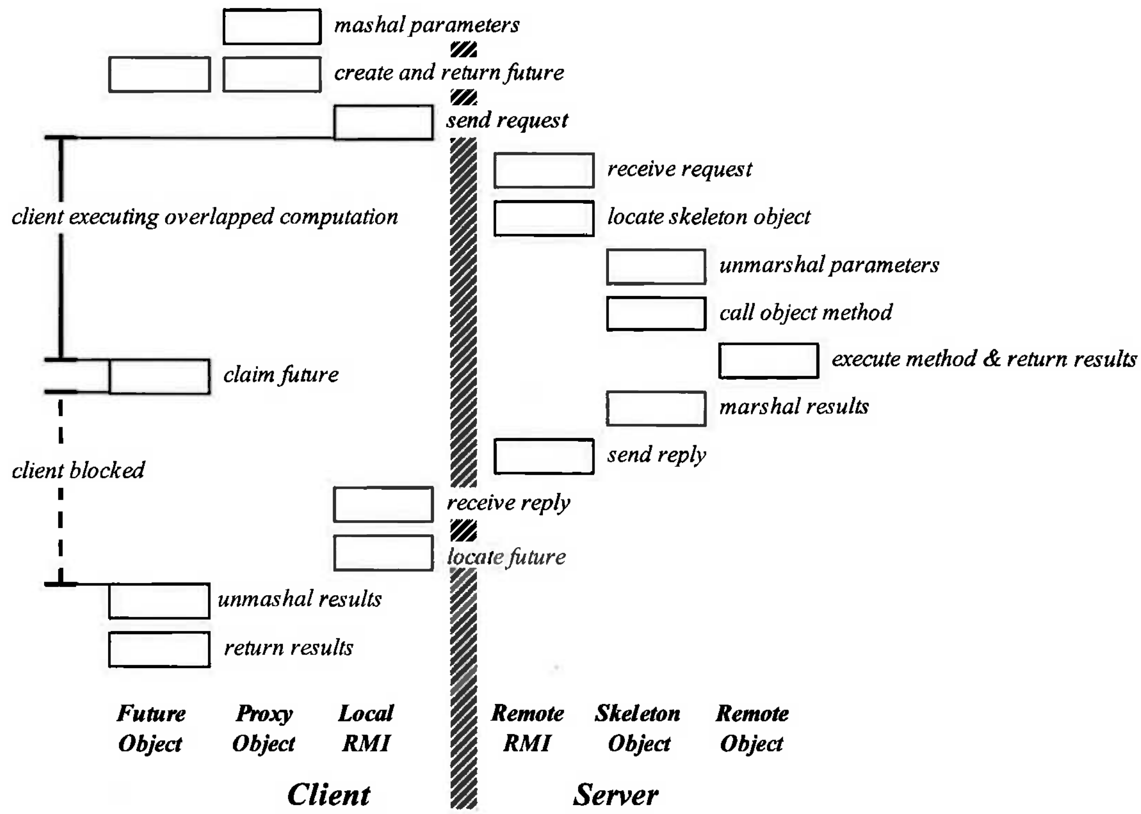

Figure 49 – Deferred Synchronous Remote Method Invocation.

of remote invocations at run-time, including the ability to split invocations into the two phases identified for futures.

CORBA Dynamic Invocation Interface (DII) is an application programmer interface (API) permitting invocations to CORBA object to be generated dynamically. In addition to this dynamicity, DII supports deferred synchronous communication, something not supported by the higher level CORBA "stubs" interface.

In respect of application issues, DII follows futures, disruption of existing source code is more severe, because of the lower level of the interface, and multiple replies to a request are not supported. The performance of DII follows that of futures exactly.

In software engineering terms, DII is less effective than most of the other mechanisms in this survey, in particular, it is less effective than the higher level CORBA "stubs" interface. Whereas communication via the stubs interface is statically typed, DII requires dynamic type checks for each communication operation, in consequence, there is no clustering of dynamic type checks. DII follows futures in respect of other software engineering issues.

6.2.3 Courier Objects

A *courier object*[84] is an object created solely for making a (single) remote call. The client creates a courier object, with a separate thread of control, and then immediately continues execution. Upon creation, the courier object performs a synchronous remote call to the server, any arguments the client wishes to pass to this call must be supplied when the courier object is created. When the call returns, its result is stored in the state of the courier. The courier object provides methods enabling other objects (such as the client) to retrieve this result and determine its availability.

[84] The term *courier object* is used without citation by Papathomas in his thesis [Papathomas 1992] and elsewhere, but the context suggests he is not the inventor, so the origin of the term is a mystery.

The courier object approach is simply an object-based approach to the implementation of multi-threaded RPC/RMI. As such, it need not be assessed independently of that mechanism.

6.2.4 Wait-by-necessity

One drawback with the futures approach is that the claim operation is explicit. Consequently, an RMI application must be modified to take advantage of the opportunity for deferred synchronisation. The Eiffel // extension of the Eiffel programming language addresses this by assigning to the language system responsibility for implicitly starting claim operations as required. This implicit initiation of claim operations is termed *wait-by-necessity*. Many alternatives are possible to support implicit claims, the approach adopted in Eiffel // is as follows:

- Only result values of reference types exhibit wait by necessity semantics, calls with result values of non-reference types exhibit synchronous semantics. Non-reference types include basic types, such as integers, and any user defined types marked by the reserved word ***expanded***.
- An additional field is added to the representation of objects of reference types. This contains a Boolean value. This indicates whether the object has been designated to hold the result of a remote call for which a reply has not yet been received. Such an object is termed an *awaited* object.
- Wherever the result of a remote call is used as the source of an assignment or as an actual argument to local method call, an object of the appropriate type is created and marked as awaited. A reference to this object is then used in the assignment, or passed as the actual argument, as appropriate, and execution continues.
- In Eiffel //, the only way to operate on an object of reference type is to call one of its methods. The Eiffel // compiler generates code so that any method call on a reference type object inspects the state of the object, and blocks if it is an awaited object.
- When a reply containing a result is received, the local object designated to hold that result is located, the state of that object is updated to hold the result value, and the object is awaited mark is removed. If the local thread of execution is blocked accessing that object, then that thread is resumed.

Notice that blocking occurs only for result returning remote calls. There is no requirement to block for remote calls that do not return results, nor is there any ability to elect to do so.

The original wait-by-necessity implementation was Eiffel //, which used a modified Eiffel compiler. However, wait-by-necessity has also been implemented in C++ [Caromel, Belloncle and Roudier 1996] and Java [Caromel and Vayssière 1998, Caromel, Klauser and Vayssière 1998], in the case of Java without compiler modifications being necessary.

Wait-by-necessity provides most of the application functionality of futures, but operates at a higher level. In particular, calls involving wait by necessity are not syntactically distinguished from fully synchronous calls, in contrast to the situation with futures. An original motivation for this construct was to facilitate continuance of use of sequential code in concurrent contexts. This goal corresponds closely with minimisation of syntactic and semantic disruption, an area in which wait-by-necessity excels.

Wait-by-necessity achieves comparable[85] critical path message counts to futures in wide area communication[86]. In respect of overlapped communication, wait-by-necessity offers slightly less support, in that only reference values are subject to wait by necessity.

[85] Since non-reference results cause blocking, the performance may be slightly worse than for futures, depending on the frequency of such results.

In respect of software engineering, the Eiffel // versions of wait-by-necessity inherits support for assertions from Eiffel. However, unlike SCOOP, there is no direct addressing of the precondition and postcondition paradoxes, although the system includes reflective facilities that could be used to provide some support.

6.2.5 SCOOP

Whereas Eiffel // is a research system, SCOOP [Meyer 1997], is a design for the addition of concurrency and distribution to commercial implementations of Eiffel. It adopts Eiffel // wait-by-necessity as an implementation alternative for reply scheduling. In addition, it extends the Eiffel language with support for concurrency and distribution in other areas, in particular, it addresses the need for modified semantics of preconditions in concurrent environments.

In respect of the issues assessed in this survey, SCOOP follows wait-by-necessity quite closely. In particular, it supports seamless deferred synchronisation with remote calls returning reference results. This main difference is that SCOOP addresses the precondition paradox via altered semantics of client preconditions.

6.2.6 Batched Futures

One restriction of futures and wait-by-necessity is that is not possible to send a future (awaited object) as an argument to another remote call. With futures, the programmer must claim the result of a future, thereby blocking if the result is not available, and then send that result, rather than the future as the argument. With wait-by-necessity, the explicit claim is avoided, but the marshalling process of the arguments to a remote call is blocked wherever it encounters an awaited object. Therefore, any remote calls including awaited objects in the transitive closure of their actual arguments are not sent until those objects are no longer awaited. Consequently, whilst a client may have several calls in progress at any given time using futures (or wait-by-necessity), there can not be any dependencies between these calls.

The ability to pass future representing the results of calls as arguments to subsequent calls permits interdependent calls to be concurrently in progress. Consider the following:

```
a := s.m(x,y);
b := s.m2(a);
```

Futures and wait-by-necessity will suspend execution of the client prior to sending the request to *m2*. The client will remain suspended until the result assigned to *a* is available. The two calls are therefore unable to be in progress concurrently.

The Batched Futures mechanism [Bogle and Liskov 1994], addresses this limitation in situations, such as the above, where a series of result dependent calls are directed to a single server. The argument to the call to *m2* is sent to the server as a future. When the server attempts to execute the call to *m2*, it encounters this future in the argument list, and substitutes the value for that future, which is the value returned by the (previous) call to *m*[87].

In the batched futures mechanism, results are either non-reference values or references to remote objects (located on the server). In contrast to wait-by-necessity and SCOOP, there is no facility for references to local objects being returned as results. However, batched futures follow wait-by-necessity in that calls returned results of non-reference types are synchronous. The innovation of

[86] To support efficient communication between processes on the same node, request sending in Eiffel // includes a two way "handshake" message exchange prior to request transmission. For wide area communication, the extra round trip entailed by this handshake would greatly harm performance, rather than improving it. However, wait-by-necessity is essentially independent of the handshake (or any particular means of request delivery), so for the purpose of performance modelling, this survey assumes that a single message request delivery is used (at least for wide area communication).

[87] Notice that there is an implicit requirement that the server execute operations in the order they are sent by the client.

batched futures is that futures may be sent back to the server responsible for producing them, without first being claimed. This has a profound impact on performance.

Before proceeding to the discussion of performance, it should be noted that the performance gaols of the batched futures implementation are rather different from those pursued in this thesis. The batched futures system was primarily designed to support communication between processes on the same machine. In this environment, the performance consideration addressed by batched futures is the overhead incurred by calls between processes. This overhead derives from context switch time and other factors. The idea leading to batched futures is that operations are batched in to a single call and the overhead is amortised across all the calls in the batch. In wide area communication, call overhead has an insignificant effect on performance, consequently batching is largely irrelevant. The relevant performance determinant is the number of operation that may be in progress concurrently. It turns out that this is determined in precisely the same way as is batching. Therefore, the discussion will continue to refer to each group of concurrently in-progress operations as a batch, in spite of the fact that they might not necessarily be batched into a single message.

For the *SSS* interaction, batched futures permit an entire R-group of operations to be in progress concurrently. Result dependencies between operations in that R-group are resolved, at the server, as the results become available. Consequently, batched futures approach[88] the heterogeneous ideal performance for this interaction. The performance for the *COSS* interaction is the same.

Modelling the multi-server interactions is somewhat problematic because the batched futures system was built for a single server environment. Consequently, it does not address the case where there are multiple servers. There are two possibilities for extension to multiple servers:

- Each batch may include operations from several servers
- Each batch may only include operations from a single server.

The second is less complex, and is assumed as the case in modelling the multi-server interactions for this mechanism. The first of these alternatives leads to a mechanism that is equivalent in performance to Mentat. This is modelled in the assessment of that mechanism.

For the *SSM* interaction, batched futures approach a critical path message count of $2r'$. In order to preserve ordering constraints between operations in adjacent R′-groups on different servers, it is necessary for the earlier group to be known to have terminated before the later group may be started. Therefore, there can be no overlapping of R′-groups, and since each R′-group requires two critical path messages, the overall critical path message count is $2r'$. The count for the *COSM* interaction is the same.

Finally, batched futures offer no additional support for the *AD* interaction. The critical path message count for this interaction remains $2a + 2$, as for futures.

6.2.7 Batched Futures with Basic Value Promises

With both wait-by-necessity and batched futures, deferred synchronous semantics are not available for results of non-reference type. This restriction is necessary because the representation of value must be augmented with a filed indicating awaited status, and this is not possible for basic values. In contrast, with futures and promises, deferred synchronous semantics are available for all result types because future (promise) values are distinguished statically (via the type system) rather than dynamically.

It is possible to combine the two approaches [Bogle and Liskov 1994], using type distinction for basic values and run-time distinction for reference values. This leads to a mechanism that entails slightly less client blocking than does the original batched futures. Concomitant with this increase in

[88] As with wait-by-necessity, the critical path message count way be adversely affected in some situations because non-reference results do not support deferred synchronisation.

performance, there is an increase in syntactic disruption, because of the reintroduction of explicit claims for basic value results.

6.2.8 Mentat and Legion

Batched futures relax the blocking requirement only for the case where the future representing a result of a remote call is used as an argument for a subsequent call to the server for that remote call. If the future were used as an argument to a call to a different server, than blocking would ensue as previously.

It is also possible to relax the blocking requirement in the general case. The Mentat [Grimshaw 1993] programming language provides this ability. The Mentat compiler constructs the dependency graph of a set of interdependent calls to any number of servers, and passes this information along with the marshalled calls. All the calls in the graph are sent to the appropriate servers immediately, and the client may then continue execution. The dependency information is used by the run-time system to forward results wherever they are needed, as they become available. Each server (and the client) is blocked only when it needs the value of an awaited result in the program graph, in which case it blocks until that result arrives. A function, *mentat_return* (), provides an early reply facility (the name stands for return to future). Calling this function does not terminate execution of the remote operation. The standard return construct can also be used. In either case, the argument passed to *mentat_return* () or **return** may itself be a result that is not yet available. In this case, dependency information is propagated rather than the value of the result, and that value is sent (from its producer) when it becomes available.

Mentat is a modified version of C++, as such, it is architecturally heterogeneous but linguistically homogeneous. The Legion system [Grimshaw *et.al.*1997] is a run-time support system providing similar facilities for (linguistically and architecturally) heterogeneous distributed systems.

Mentat is quite effective in addressing most of the application issues. Replies are supported, with reply matching being performed by the system and reply synchronisation being an explicit/implicit hybrid. The *mentat_return*() function provides an early reply mechanism. Furthermore, the ability to send the result of an incomplete remote operation as a return value supports request forwarding. Coordinated termination is supported, although not automatic, and one-way requests are supported because request submission is non-blocking, as with futures. Mentat is oriented towards client/server communication, peer symmetry is not supported and reply entailment reaches only the simple client/server level.

However, Mentat (and also Legion) is not designed to integrate into existing programming languages. Instead, complex modifications are needed to the implementation language compilers in order to integrate them with the middleware. The Mentat compiler is a modified C++ compiler, the modifications are of the following kinds:

- Addition of analysis functions for determination of dependency graphs for sets of remote calls.
- Loop unrolling, to support construction of dependency graphs for dependencies between operations executed by different iterations of a given loop or by different loops.
- In line transformation of source code to include calls to marshalling, message transmission graph construction and result claim functions.
- Modifications to support other Mentat language features, incidental to the topic of this thesis.

The last of these is irrelevant to middleware integration (the unnecessary language features could simply be removed). Marshalling, message transmission and result claim can all be supported via a combined library and stub compiler approach, as demonstrated by other mechanisms. However, supporting the requirement for graph analysis and construction and for loop unrolling is more problematic. Assistance from the implementation language compiler (or a pre-compiler) is necessary[89] for high level support for communication with this mechanism. Hence, Mentat and Legion avoid syntactic and semantic disruption of existing source code by requiring new compilers for that source. This is a very significant implementation burden.

On the software engineering side, Mentat follows futures exactly. There are no fundamental incompatibilities between Mentat and support for assertions, although the language (being based on C++) does not support them.

The big advantage of Mentat is in terms of performance. For the *SSS* interaction, Mentat follows Batched Futures (with basic value Promises) in achieving the heterogeneous ideal message count. Mentat also achieve the heterogeneous ideal for the *SSM* interaction, unlike Batched Futures. This is possible because the dependency graph propagation makes possible forwarding of results directly between servers[90]. The situation is similar for the *COSS* and *COSM* interactions: Mentat achieves the heterogenous ideal for both. Furthermore, because of the support for request forwarding, Mentat achieves the ideal case for the *AD* interaction. Finally, because all remote operation requests are non-blocking, Mentat achieves the level of least constraint on overlapped execution.

6.3 One-way Message Passing

All of the mechanisms described above are developed within the context of request/reply communication, since this is the straightforward mapping of method call to distributed systems. Method call is certainly the mainstream approach to inter-object communication, however it is not the only approach. This section discusses mechanisms in which each inter-object communication is a one-way transfer. Assessment of these mechanisms is summarised in Table 18.

6.3.1 Distributed Actors

Actors [Agha 1986], are an approach to concurrent object oriented programming in which communication is through one-way message passing at the conceptual level. Communication facilities for Actors based distributed systems are based on one-way message passing at the implementation level. Consequently, the communication patterns in distributed Actor systems lack the request/reply structure exhibited by most other distributed object systems. The steps taken for a remote method invocation in an Actors distributed system are shown in Figure 50.

The most important difference between one-way message passing and the mechanisms discussed previously is the absence of replies and request/reply structure. This is positive in one sense: peer interactions need not be encoded into client/server interactions, and one-way requests are automatically supported (indeed, they are the only kind of requests that are supported). However, in most senses, the lack of a system-supported notion of replies is a disadvantage, in that developers are completely responsible for the structure of communication, and are not assisted in achieving coordinated termination. Furthermore, although one-way requests can be seamlessly integrated into implementation languages via proxies and skeletons, the call-return paradigm of sequential programming cannot be mapped to distributed systems in the absence of replies. Consequently, developers familiar with sequential object-oriented programming are forced to adopt a paradigm other than that with which they are familiar.

Actors are compatible with static type checking and with clustering of dynamic type checks. Assertion based contracts are not supported in any Actors system known to the author (whether

[89] See the Legion Developer Reference Manual [Legion 1998]. There is a stub compiler for Legion (the "object interface compiler") that permits the use of a standard C++ compilation process, however this has strict RPC/RMI semantics; there is no graph construction, no deferred synchronisation at all.

[90] Additional programmer work, possibly involving the propagation of artificial results, may be needed to ensure ordering constraints between operations on different servers are preserved, if that is desired.

Capability	**Distributed Actors**	**Synchronizers**	**Asynchronous Reply**
Replies	False	True	
Early Reply	Not Applicable	True	False
Reply Matching		No support (1/3)	
Reply Synchronisation		Asynchronous (1/3)	
Reply Kinds	All		
Coordinated Termination	No support (1/3)	Supported (2/3)	
Syntactic Disruption	Significant (1/3)		
Semantic Disruption			
One-way Requests	True		
Request Forwarding	Not Applicable	Linguistic (3/4)	No support (1/4)
Peer Symmetry	True		
Reply Entailment	No support (1/3)	Extended (3/3)	
Reply Races	Not Applicable.	All values (3/3)	
Multiple Replies		True	
Interaction Structure		General (4/4)	Fixed (1/4)
Linguistic Heterogeneity	True		

Capability	**Distributed Actors**	**Synchronizers**	**Asynchronous Reply**
Type Checking Timing	Static (3/3)		
Type Checking Clustering	True		
Assertion Coverage	Not Applicable		
Intrinsic Assertion Specification			
Inseparable Assertion Specification			
Declarative Assertion Specification			
Early Assertion Checking			
Precondition Synchronisation			
Asynchronous Contract Completion			
Reply Specification Coverage	No support (1/3)	Extended (3/3)	
Multiple Reply Entailment	false	true	false
Interaction Structure Specification	Fixed (1/4)	General (4/4)	Fixed (1/4)
Intrinsic Reply Specification	Not Applicable	False	True
Inseparable Reply Specification			
Declarative Reply Specification		True	
Positive Reply Specification			
Reply Sending Enforcement			
Illegal Reply Handling		Rejection (3/4)	Prevention (4/4)
Illegal Reply Rejection Timing		Dynamic (1/2)	Not Applicable.
Extra Reply Handling		Rejection (3/4)	Prevention (4/4)
Extra Reply Rejection Timing		Dynamic (1/2)	Not Applicable.
Asynchronous Method Call Exception Handling	False		True
Exception Handling Contexts	Not Applicable		Object (1/4)
Exception Handling Flexibility			True

Interaction	**Asynchronous Reply**
$\mathrm{I}_{SSS}(f,r,o,n,m)$	$2n$
$\mathrm{I}_{SSM}(f,r,o,n,m,s,r',o')$	$2n$
$\mathrm{I}_{AD}(a)$	$a+2$
$\mathrm{I}_{COSS}(f,r,o,i,n,m)$	$2n$
$\mathrm{I}_{COSM}(f,r,o,i,n,m,s,r',o',i',ir')$	$2n$

Overlapped Computation	(7/7)
Request Transmission Policy	Eager (3/3)
Reply Transmission Policy	

Table 18 – Assessment Checklist– One Way Message Passing Mechanisms.

distributed or otherwise), but are compatible with such support. The absence of language support for replies leads to an absence of reply contracting facilities, a notable weakness in comparison to other mechanisms in this survey. Finally, because all communication is one-way, there is no exception signalling and handling facility.

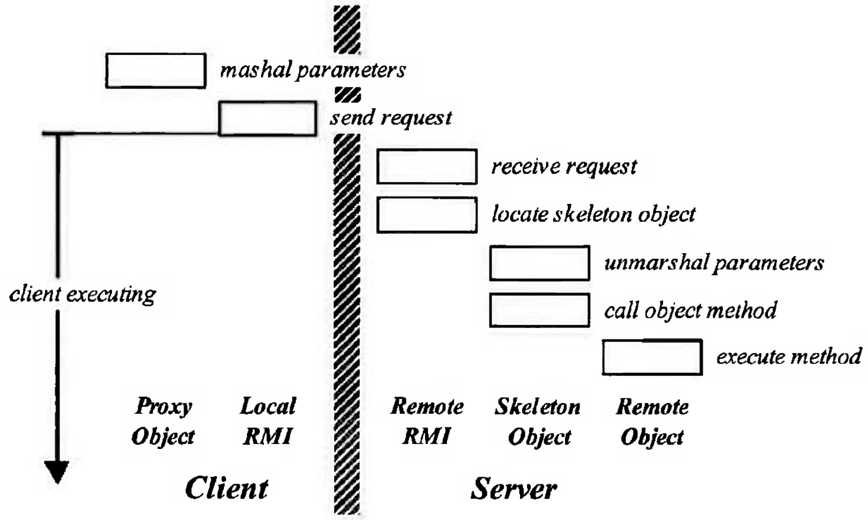

Figure 50 – One Way Remote Method Invocation.

The framework for performance assessment is somewhat ill suited to modelling the performance of systems in which there is no notion the result of a method call. Given the absence of such facilities, developers implementing the interactions in the performance-modelling framework need to develop reply/result infrastructure at the application level. This infrastructure could reach an extremely sophisticated level of elaboration, for example it could provide the performance exhibited by Mentat. However, the assessment of such infrastructure would apply to application facilities rather than facilities provided by the communication mechanism. Consequently, there is no assessment of the performance of this mechanism – the actual performance will depend on the implementation effort expended by application programmers. Note, however, that the performance for each interaction is bounded by the heterogeneous ideal.

6.3.2 Supervised Actors

Synchronizers [Frølund and Agha 1993], are supervisory agents in Actor systems which constrain the communication exhibited by a set of Actors to follow patterns determined by the synchronizer. Rather than being confined to a predetermined structure, as in request/reply communication, programmers using synchronizers can specify structures that are most appropriate for their application. The Synchronzers mechanism is really a specific case of the general concept of an extrinsic supervisor for a group of communicating objects (Actors). The authors state that the system they describe is intended to illustrate this general concept rather than being a fully developed facility. Consequently, this section assesses the general approach of supervised one-way message passing, rather than the specific example of Synchronizers.

Supervisors such as Synchronizers allow the constraint of communication patterns between Actors. For example, a server and a collection of clients could be constrained to follows request/reply communication structures. This gives support to the notion of reply, although individual Actors continue to view communication a series of one-way message transmissions. In particular, there is no support for reply matching or reply synchronisation but early reply is intrinsically supported. Similarly, a supervisor could impose constraints to support coordinated termination[91]. As with the unsupervised Actors approach, the integration of supervised Actors middleware into most implementation languages[92] is non-seamless, because of the impedance mismatch between one-way

[91] In order to provide effective support for coordinated termination, the supervisor would need to specify temporal constraints on the transmission of reply messages, in addition to structural constraints. The Synchronizers mechanism includes only structural constraints, but there is no reason it could not be extended to include temporal constraints.

[92] Integration into Actors-based implementation languages would be straightforward, but such languages are a very small minority.

```
class SERVER
  sm1(p1:INTEGER) is ...
  sm2(p1:STRING) : INTEGER is ...
  sm3(p1,p2:STRING) : callback(INTEGER,STRING) : INTEGER is ...
end

class CLIENT
  cm1 is ...
  cm2(p1:INTEGER) is ...
  cm3(p1:INTEGER;p2:STRING) : INTEGER
end

s.sm1(34) =: cm1       -- Call remote sm1 method, speicify local
                       -- cm1 method as reply handler.
s.sm1(45)              -- One way request to remote sm1 method,
                       -- speicify no local method as reply handler.
s.sm2("foo") =: cm2    -- Call remote sm2 method, speicify local cm2
                       -- method as reply handler.
s.sm3(67,"foo") =: cm3-- Call remote sm3 method, speicify local cm3
                       -- method as reply handler.
```

Figure 51 – An Example using Asynchronous Reply.

communication and method call-return. However, integration of the supervisors themselves is not problematic: IDL supervisor definitions are used to generate supervisor implementations that are used only internally within the middleware. One-way requests are intrinsically supported by one-way message passing, and supervisor constraints expressing request forwarding could be supported. The great advantage of this approach is that it allows communication to be structured, but does not impose any particular preconceived structure; request/reply supervisors are merely one class of possible structural supervisors. Supervisors are equally able to support structuring of peer interactions in which peer symmetry is maintained. Supervisors also support extended request/reply sequences in both the client/server and peer contexts.

Supervisors constraints are intrinsically specifications of communication structures. The most important examples of such structures are request/reply structures (in both client/server and peer interactions). However, unlike many other mechanisms, supervisors are not limited to request/reply structures. Furthermore, supervisors can specify request/reply structures that go beyond the simple case of request/reply in client/server interactions. Supervisors are separate from the Actors they supervise, consequently the communication structure specification is neither inseparable nor intrinsic. In the specific case of Synchronizers, the specification is a hybrid of operational and declarative forms, although a purely declarative form could be substituted. Specification is also positive in form. In order to support enforcement of reply sending. Supervisor specifications need to support temporal constraints, Synchronizers do not, but could be extended to do so. Structural constraints, such as those supported by Synchronizers, are sufficient to enforce reply validity, but the enforcement is necessarily dynamic. Finally, there is no system support for exception handling: exceptions would need to be indicated by ordinary one-way messages.

As with unsupervised Actors, attainment of a particular level of performance is primarily the responsibility of application programmers rather than of the system. Consequently, modelling of the performance of this mechanism is omitted.

6.3.3 Asynchronous Reply

Asynchronous reply is a technique for constructing request/reply interactions by composition of one-way requests. All messages are one-way requests, however, in addition to the request parameters, each message may optionally specify a *reply handler*. This indicates a method of the caller that the target of the request is to call, passing the result of the (first) request as parameters. The message delivering the results to the delegate is itself a request, so it, too can include a delegate method to receive the results computed by the delegate. Consequently, asynchronous reply can support replies to replies as well as replies to requests.

The specification of a server object using an invented IDL syntax (derived from Eiffel) supporting asynchronous reply is shown in Figure 51. The specification of a client to interact with this server is also shown in the figure. Finally, the figure shows some examples of client code for interacting with a server instance denoted *s*.

One major drawback with this approach is that it requires support from the compiler(s) of the client side implementation languages. This is required to suport association of (client side) reply handler methods and server side methods generating those replies. For a strongly typed implementation language, the compiler must perform type checking on each association of a reply handler method with a request. Even if the implementation language is untyped, there is still a need to be able to refer methods s language objects something which is not supported in most object oriented implmentation languages[93].

Asynchronous reply effectively introduces the concept of reply into one-way message passing. However, support for replies is at a lower level than was the case for most of the RMI and deferred synchronous mechanisms. In particular, there is so system support for reply matching, nor for synchronisation with the arrival of replies. Return request (reply) messages are sent only after return of the operation triggered by the original request, although there is no fundamental incompatibility with an extra mechanism for early reply. Similarly, request forwarding is not supported, although the basic mechanism could be extended to support it. The main application advantage of this approach is that it supports replies whilst permitting peer symmetry, and furthermore, it supports extended reply entailment in that replies are themselves requests and may consequently have further replies associated. There is, however, at most one reply corresponding to a given message.

Support for software engineering is quite effective. The mechanism is compatible with static type checking, and also with assertions. Support for reply contract specification is strong: the mechanism supports specification of extended request/reply sequences. It is intrinsic, inseparable, declarative and positive in form. Both reply sending and reply validity contract clauses are enforced, in fact the structure of the system statically prevent sending of invalid replies. The mechanism may be extended to support asynchronous exceptions by allowing each remote call to specify an exception handler method in addition to a reply handler.

The performance of this mechanism is equivalent to that of futures, in that a collection of order dependent requests to a given server may be in progress concurrently and there is no specific support for request forwarding.

6.3.4 Delegated Reply

With asynchronous reply, the result of a request is delivered via a method call to a specified method of the originator of that request. An extension to this facility is to allow the originator to specify that some other object is to be the target of the return request. This object is referred to as the *delegate* for the request. A two way communication using this approach is illustrated in Figure 52. Typically, the delegate is located in the same process as the originator, but executing concurrently with it. When

[93] C++ supports the concept of pointer-to-member-function, which is a reference to a method, with a particular signature, and which is a member of a particular class (or a derived subtype thereof). Consequently, one might attempt to support reply handlers within C++ by adding a extra parameter for the reply handler to each parameter list in the server proxy. This parameter would be of a pointer-to-member type with appropriate parameter types. However, the type of a pointer-to-member-function includes the type of the class containing the member function, so only clients conforming to the type of that class could supply appropriate reply handlers. This constraint is prohibitively onerous. Consequently, this mechanism is unsuitable for the specification of reply handlers as parameters.

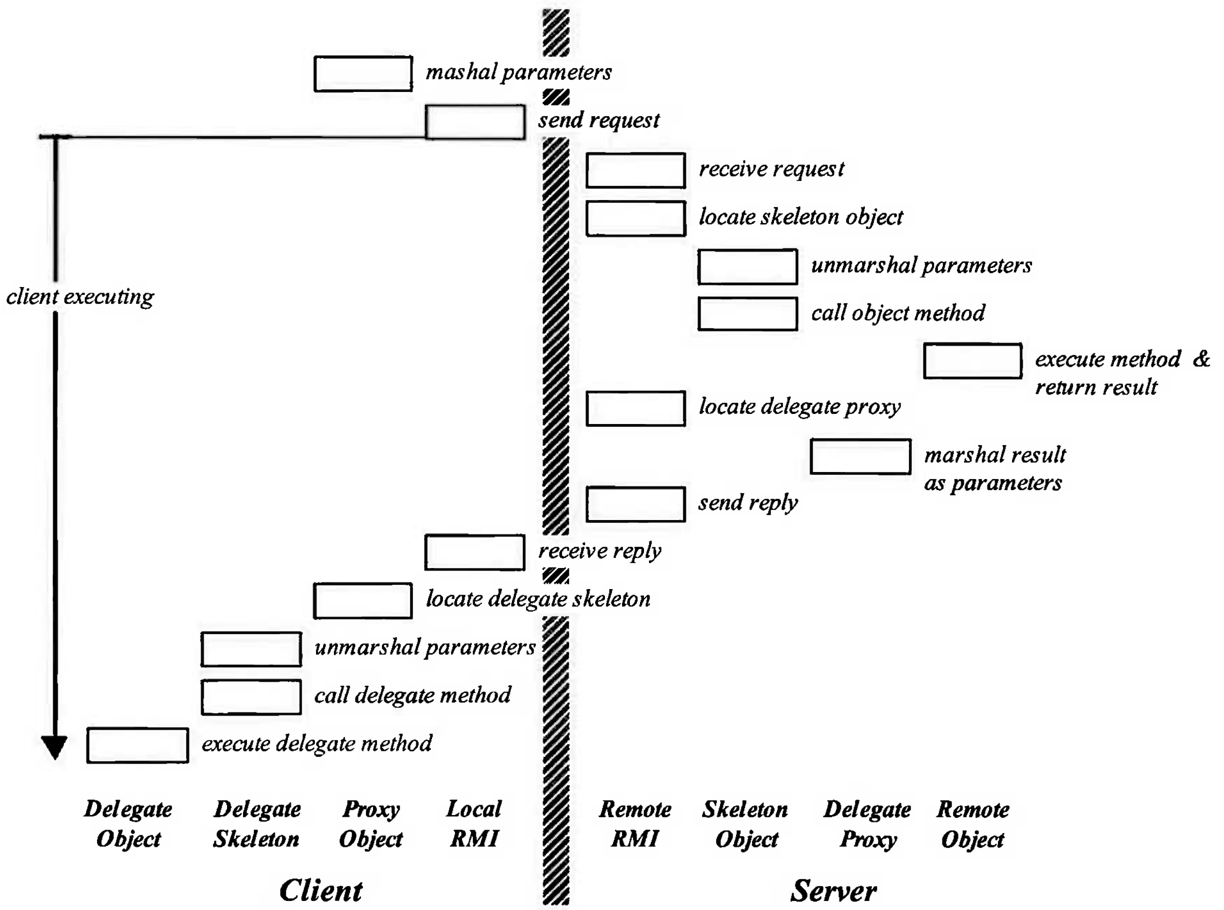

Figure 52 – Remote Method Invocation with Delegated Reply.

called, the reply handler simply saves the result in an instance variable of the delegate. The originator can synchronously retrieve the result of its request by making a local method call on the delegate.

The approach described above is equivalent to the futures mechanism.. However, a reply redirection mechanism can also be used in ways that are not equivalent to futures. For example, the reply handling method could do something in addition to (or instead of) saving the result. The ABCL language provided delegated reply in addition to futures. However, uses for this additional flexibility are not common, there fore this approach is not assessed independently of futures.

6.4 Reflective Approaches

Reflective approaches enable advanced programmers to access the communication mechanism below the standard language level interface and thereby avoid some or the limitations of that interface, at the cost of additional complexity and difficulty of use.

In Eiffel //, Caromel provides extensive reflective capabilities enabling access to the implementation of the communication mechanism. In particular, it is possible to reify the data structures used for communication and then manipulate them as standard Eiffel data structures. It is also possible to modify the marshalling procedure. The Java RMI, serialization (marshalling) and reflection sub-systems support similar capabilities.

Through use of these facilities, it is relatively straightforward [Caromel and Detmold 1995] to extend the Eiffel // wait by necessity mechanism with the following:

- Seamless batched futures functionality.
- Request forwarding, via a library interface.
- An early reply mechanism, similarly via a library interface.
- Reply races, in which only the winning reply value is accessible, also via a library mechanism.

Interaction	Mechanisms	
	Emerald/DOWL (Object Migration)	**Stored Procedures/RX**
$I_{SSS}(f,r,o,n,m)$	$2f$	
$I_{SSM}(f,r,o,n,m,s,r',o')$	$f+s$	$2s$
$I_{AD}(a)$	$2a+2$	
$I_{COSS}(f,r,o,i,n,m,s)$	$2f$	
$I_{COSM}(f,r,o,i,n,m,s,r',o',i',ir')$	$f+s$	$2s$
Overlapped Computation	Not applicable	(1/7)

Table 19 – Performance Assessment Checklist– Migration Based Approaches.

The addition of Mentat-like functionality would be more challenging because of the lack of compiler support for pre-computing the dependency graph. In the absence of such support, a reflective approach would need to compute the dependency graph as requests are submitted. It would then forward this dependency information to relevant servers.

The reflective approaches are powerful, and capable of supporting any of the application functionality and performance levels attained by mechanisms described above. However, they are not of a sufficiently high-level to provide effective support for the model software engineering adopted in this thesis, and so are rejected as unsuitable and not further assessed.

6.5 Migration Based Approaches

The approaches discussed thus far have assumed that heterogeneity requirements preclude the ability to migrate executable code across the network in order to optimise communication performance. The application component of the framework also makes this assumption. However, the performance advantages of code migration approaches can be considerable, so they are discussed in this section. In this survey, these approaches are assessed in respect of performance only, without assessment of the other component of the framework. The assessment is summarised in Table 19.

6.5.1 Batched Control Structures

Consider a client that executes the following code involving a series of dependent calls to a remote object:

```
for x in 1..5 loop
  o := s.get-next(o)
end
```

In Mentat, the loop is unrolled by the compiler into a basic block with five calls. With batched futures, the five calls in the execution of the loop are batched accumulatively as they are submitted by the client. In both cases, the client continues through the loop without blocking.

However, in both of these cases, each of the five calls needs to be marshalled and unmarshalled independently. Consequently, the cost of marshalling and unmarshalling is proportional to the number of calls executed. Batched control structures [Zondervan 1995, Liskov *et.al.* 1996], are a development of batched futures which transmits the loop itself, rather than the calls resulting from its execution. With this technique, the cost of marshalling and unmarshalling is proportional to the number of remote call in the client source code. In the example above, this is one call rather than five for batched futures and Mentat. In addition to ***for*** and ***while*** loops, ***if*** statements can also be batched.

Batched control structures enhance performance significantly for remote calls between processes on the same machine and on different machines in the same local area network. This is achieved as a result of the reduced marshalling and unmarshalling cost. However, as discussed in the previous chapter, the time taken in marshalling and unmarshalling is dwarfed by the communication delay in worldwide systems. The sensitivity to communication delay for batched control structures is the same as that for batched futures with basic value promises, as is the effect on client blocking. Therefore, there is little point in this survey assessing the performance of batched control structures independent of that mechanism.

In addition, batched control structures require a limited form of executable code migration. In this case, the executable code is a mini-language defined and interpreted by the middleware, so the mechanism escapes the usual problems raised by code migration in linguistically heterogeneous systems. However, the client programmer must be able to provide batches of control structures to the middleware. This precludes a simple binding from the client programming language to the middleware. Instead, one of the following alternatives must be chosen:

- The compiler for the client programming language forms remote operations into batches and submits the batches to the middleware. Modifying the compiler to support the middleware is an unrealistic burden to impose on a client programming language implementation.
- The programmer supplies batches via library calls, thereby losing the benefits of compiler supported[94] standard control structures in code that performs remote operations.

Neither of these alternatives is satisfactory.

6.5.2 Stored Procedures

Many relational databases systems support the concept of stored procedures. These are procedures and functions, written in some database system specific programming language with embedded SQL, that are uploaded to the database server and made available to be called by clients. The main advantage of this approach in respect of distributed systems performance is that a given stored procedure can contain multiple SQL statements. Therefore, client programmers can execute a complex series of operations by a single call to the appropriate procedure, rather than a series of functionally dependent SQL statements.

This approach is currently not applicable linguistically heterogeneous systems, because the procedures must be written in a language determined by the database system. In the future, stored procedures will be incorporated into the SQL standard, making the approach suitable for access to a heterogeneous set of database servers, though still not for general distributed systems use.

For the *SSS* interaction, this mechanism attains the best possible performance. An entire F-group of operations, including any functional (and lower order) dependencies between those operations, can be sent to the server in the form of a stored procedure, taking a single critical path message. These operations are then executed by the server, and their results returned, taking a second critical path message. The overall interaction message count is hence $2f$, where f is the number of F-groups. The performance for the *COSS* interaction similarly attains the ideal.

For the *SSM* interaction, the ideal is not attained. The critical path message count is two messages for each S-group in the interaction. This is necessary because there is no means for inter-server request chaining to preserve order dependencies between operations on different servers.

The stored procedures mechanism does not include any specific support for request forwarding. Consequently, the performance for this interaction is the same as for RMI.

Finally, calls of stored procedures block the client. Consequently, with a single threaded client, there is no ability for more than one stored procedure to be in progress at one time. Therefore, it is not possible to execute order independent operations on different server concurrently, and the performance if the *COSM* interaction is the same as for *COSS*. The blocking nature of stored procedure calls also prevents the client from n overlapping computation with communication.

[94] The batched control structures implementation uses pre-processor macros to embed batching information within C++ programs. This leads to several types of syntax errors that could only be detected at run-time.

```
old := Self.$LOCATION
Self.$LOCATION := server.$LOCATION   -- one critical path message
… collection of calls to methods of the server …
Self.$LOCATION := old                -- one critical path message
```

Figure 53 – Expediting execution of an F-group using $LOCATION.

```
old := Self.$LOCATION
Self.$LOCATION := server1.$LOCATION  -- one critical path message
… collection of calls to methods of server1 …
Self.$LOCATION := server2.$LOCATION  -- one critical path message
… collection of calls to methods of server2 …
Self.$LOCATION := old                -- one critical path message
```

Figure 54 – Inter-server Request Chaining using $LOCATION.

6.5.3 RX

In addition to relational database servers, other distributed systems elements could support a stored procedures interface. One example is the RX proposal [Dearle, Rosenberg and Vaughan 1991] for communication between implementations of the persistent programming language Napier88. Napier88 supports first class procedures, so it is relatively trivial to transmit procedure objects to a server for later (or immediate) execution. Once again, however, this mechanism requires that procedures be defined in the language[95] of the server, so is not applicable for linguistically heterogeneous systems. In addition, the relevant performance characteristics of this mechanism do not differ from those of stored procedure so there is no need for a separate assessment of this mechanism.

6.5.4 Emerald and DOWL

The Emerald programming language [Black *et.al.* 1986], later followed by DOWL [Achauer 1993], support the construction of object based distributed systems in architecturally heterogeneous environments. These systems contain extensive support for object migration, in particular:

- Objects passed as parameters to calls to remote objects are automatically migrated to the location of the remote object. This mechanism is termed *call-by-move* and is intended to reduce the latency of operations on the parameter objects (and therefore the time taken to execute the call) whilst preserving the (single copy) semantics of passing objects by reference.
- All objects have a special attribute, called *$LOCATION.* Assignment of one objects *$LOCATION* attribute with the value of that of another object explicitly migrates the second object to the location of the first. Communication between the two objects can thereafter avoid network transmissions and the consequent delays.

However, migration of an object implies migration of the code of its methods. As pointed out previously, this rules out this mechanism from use in middleware for linguistically heterogenous systems.

As with stored procedures, this mechanism can attain the ideal performance levels for the *SSS* and *COSS* interactions. One way to attain this performance is to use the $LOCATION property to expedite the execution of each F-group as shown in Figure 53[96]. This level of performance for these

[95] In RX, this is also the language of the entire system, that is, Napier88.

[96] Note that full dependencies can only be traversed on the original location of the client object, so the fact that the client object has migrated does not allow full dependencies to be traversed at the server end. Full dependencies are intended to model dependencies between operation that involve interaction with the environment of the client object, for example interaction with the client's user.

interactions can also be attained by using call-by-move, passing the client as a parameter to a call on the server.

In contrast to stored procedures, the technique based on assignment to the client's *$LOCATION* property can also attain the ideal performance for the *SSM* interaction. An example for an F-group including two S-groups is shown in Figure 54. However, attaining the ideal performance for the *COSM* interaction would involve splitting up the client to execute independent operations on different servers concurrently. Consequently the assessed performance for this interaction is the same as for *SSM*, and marginally worse than the ideal.

This mechanism converts remote calls into local calls. This is achieved by migrating the client to the location of the target. The consequence of this is that the client is never blocked awaiting the results of communication, and therefore there is no potential for computation to be overlapped with communication.

Chapter Seven

Entitlements and Responsibilities

The facilities by which remote operation results are managed are an essential differentiating characteristic of communication mechanisms for wide area distributed object systems. This chapter presents a new mechanism for communication in such systems and consequently, a new approach to result management.

The first mechanism discussed is based of the concept of clients obtaining entitlements to the results of operations that they have previously requested to execute, as discussed in Section 7.2. This mechanism, *Entitlements*, is presented in Section 7.3 and is assessed according to the framework of Chapter Five in Section 7.4 This is developed into a more powerful mechanism, *Responsibilities* [Detmold and Oudshoorn 1996a,1996b], presented in Section 7.5. This second mechanism is assessed in Section 7.6

7.1 General Context of Middleware System

This chapter primarily concerns the development of facilities to manage results of asynchronous remote method invocations. However, this development occurs within the context of a middleware system. Some properties of this system are relevant to the development of the result management facilities, including:

- Following Java RMI [Sun 1997a], there is a special class (interface), called *REMOTE*. Instances of this class are passed by reference in messages. Objects that are not instances of this class are passed by value (closure copy).
- For each class implementing *REMOTE*, there is a corresponding proxy class that is conformant to the type of the class. Typically, this means the proxy is a sub-type of the type of the class.
- References to objects of classes implementing *REMOTE* are represented directly in the node where the object is resident and by proxies in other nodes.
- When a local object of a class implementing *REMOTE* is marshalled, it is marshalled as an object of the corresponding proxy class.
- Proxies unmarshalled at the node to which they refer are replaced by a local reference to the referenced object.
- Proxies unmarshalled at any other node continue to be represented as proxies.
- The semantics of passing objects by value are to preserve sharing relationships between objects passed in a given request. Therefore, if single request refers to a given (non-*REMOTE*) object more than once, all references within that request refer to a single shared object. However, if different requests are passed references to a given local object, then each of those requests has an independent copy of the object.

7.2 The Concept of Entitlements to Results

A client that calls a remote operation establishes a contract with the server that, amongst other things, entitles the client to receive the result of the remote operation. Probably the most important step in improving the flexibility of result management facilities is to separate result management and remote operation call and return, by making entitlements to results available at the language level in some form. This step is critical in that it permits the introduction of asynchrony in remote communication without unnecessarily requiring programmers to face the perils of multi-threading. This thesis describes a language level mechanism called *Entitlements* that partially exposes a separation of result management and operations call/return at the language level.

7.2.1 A New Construct?

It is important to state that, as a general concept, *Entitlements* are not new, following from similar concepts such as *Promises* [Liskov and Shrira 1988], *Futures* [Walker, Floyd and Neves 1990] and *wait-by-necessity* [Caromel 1993].

As will be seen later, *Entitlements* are related to *Responsibilities* both in terms of sub-typing (a Responsibility type is a subtype of the corresponding Entitlement type) and in terms of inheritance (a Responsibility class extends the corresponding *Entitlement* class). This being the case, *Entitlements* can be viewed as a stepping-stone on the path to *Responsibilities*. Existing constructs do not quite fit in this role.

The construct is given the new name *Entitlement*, rather than the traditional *Promise* or *Future*, in order to draw attention to the fact that it is only the client side of the contract that is of interest here. In English usage, though not in fact in the communication mechanisms, the terms *Promise* and *Future* connote both supplier and receiver parts of a contract for result supply, whereas entitlement refers only to the client side. This rather subtle distinction is important for clear-minded consideration of *Responsibilities* later in this chapter.

Finally, as explained in the discussion of design gaols for Entitlements, there are differences in detail between Entitlements and previous constructs. These differences, taken with the issues raised above, justify viewing Entitlements as a distinct and (partially) novel mechanism.

7.2.2 Terminology

In addition to the term *entitlement*, several other terms are used in the discussion of entitlements, including the following:

- An operation requiring the actual value of the result associated with an entitlement is referred to as a *demand* for that entitlement.
- The system node issuing a remote method call leading to the creation of an entitlement associated with the result of that call is referred to as the *primary demander* for that entitlement.
- The arrival of the actual result associated with an entitlement at the primary demander for that entitlement is termed *fulfilment* of that entitlement.
- Sending of the result value that will be used in fulfilment of an entitlement is termed *supply* of that entitlement.
- The node supplying the result value for an entitlement is termed the *responsible supplier* for that entitlement.

Additional terminology relevant only to a part of the discussion is introduced as needed.

7.2.3 Design Goals for Entitlements

The design of a mechanism for managing entitlements to remote operation results is challenging. The following goals are identified:

- *Automatic synchronisation* – synchronisation should occur automatically when an attempt is made to use a result that is not yet available. This goal is the essence of Caromel's wait-by-necessity principle [Caromel 1993], but distinguishes Entitlements from mechanism such as

Promises [Liskov and Shrira 1988] and Futures [Walker, Floyd and Neves 1990], which require an explicit (claim) operation to effect synchronisation (and retrieval of the result).

- *Minimal synchronisation* – synchronisation should be minimal; it should apply only to operations that access the actual value of the result, not to assignments or parameter transmission. This distinguishes Entitlements from Promises [Liskov and Shrira 1988], Futures [Walker, Floyd and Neves 1990] and wait-by-necessity [Caromel 1993], all of which force a synchronisation when a result is transmitted in a subsequent remote call. In pursuing this goal, Entitlements are also distinguished from Batched Futures [Bogle and Liskov 1994]. Whilst Batched Futures avoid forcing a synchronisation when a result is transmitted in a subsequent method call to the server that produced that result, in the Batched Futures mechanism is tied to a single server and a synchronisation is necessary when a result is transmitted to some other server. This restriction is lifted with Entitlements.
- *Reuse of code from existing synchronous systems* – the majority of existing program code is both sequential and designed for centralised rather than for distributed systems. In such code, operations are fully synchronous. The entitlements mechanism should allow reuse of this code without modification but with asynchrony (and hence concurrency) introduced by the mechanism to enhance performance in the distributed context. This also is one of Caromel's primary design goals.
- *Type orthogonality* – the entitlements mechanism should be orthogonal to the type system. That is, it should support all kinds of types equally, rather than being restricted to certain kinds of types. This goal is shared with Promises [Liskov and Shrira 1988], Futures [Walker, Floyd and Neves 1990] and Batched Futures [Bogle and Liskov 1994], but distinguishes Entitlements from wait-by-necessity [Caromel 1993], which supports only reference types (class objects) as results with wait-by-necessity semantics.
- *Contract support* – the entitlements mechanism should support design by contract.

Each of these is discussed in more detail below.

7.2.3.1 Automatic Synchronisation

In order to support automatic synchronisation at the point of use, the entitlements mechanism must be interposed between the user code and the result that is being used.

The most common use of a result is the application of some user-defined operation to it. In addition to user defined operations, a programming language supports several system level operations on objects or values.

The kinds of languages of interest in this thesis are object-oriented, strongly-typed and statically typed with a dynamic typing facility for downward traversal of type hierarchies. Examples of such languages include Eiffel, Java, C++ and Theta [Liskov *et.al.* 1995]. System level operations in these languages include:

- Operations on the actual type of the object. For example, the run-time type information (RTTI) facilities in C++, the reverse assignment attempt in Eiffel and the dynamic typing constructs in Java.
- Deallocation of the result.
- Dereference of the result.
- Comparison of the result to another object. Typically this is an identity comparison for reference values and a direct comparison for non reference values.
- Raising the result as an exception and subsequent matching and handling of this exception.

Each of these operations applies to the actual value of the result, so it is necessary for these operations to be synchronised with the result's arrival in the same way as user-defined operations.

7.2.3.2 Minimal Synchronisation

Synchronisation should occur only when a result is used in such a way as to require the actual value of that result. It should not occur in the following cases:

- A result is assigned to a variable.
- A result is passed as a parameter to a local or remote method call.
- The result is returned as the result of another local or remote call.

It is particularly important that synchronisation does not occur when the result is passed as a parameter to a remote call, as this allows method calls that are result-interdependent to be concurrently in progress, leading to a reduction in critical path message count, the most important performance determinant according to the ranking given in Section 5.3.4

7.2.3.3 Reuse of Code from Existing Synchronous Systems

The entitlements mechanism should operate on the existing base of user source code without requiring large-scale modifications at the syntactic level. It should also preserve the semantics of the sequential system as much as possible whilst conferring the advantages of minimal synchronisation on that code. The interposition of automatic synchronisation so that operations applied to an entitlement are synchronised with the arrival of the result fulfilling that entitlement goes part of the way towards attaining this goal.

In addition to imposing automatic synchronisation, the entitlements system must hide other differences due to asynchrony and distribution from the user. The issues to be dealt with are:

- The goal of minimal synchronisation requires that values may be transmitted to other nodes in entitlement form prior to the availability of the result at the transmitting node. The system must ensure that when the result is finally available, it is accessible at the receiving node(s).
- Furthermore, a node receiving an entitlement over the network may receive a given entitlement more than once, possibly in different request or reply messages. The system must ensure that the referential integrity relationships between the results associated with these copies of the entitlement are the same as the referential integrity relationships that would exist if the actual result had been transmitted rather than the entitlement. There are three cases for referential integrity relationships between values:
  - Values that are references to objects that implement the *REMOTE* interface. These have system-wide referential integrity.
  - Values that are references to objects that do not implement the *REMOTE* interface. A new copy of the closure of each such object is created for each message in which it is passed, hence these values only preserve referential integrity within a given remote call and within local calls.
  - Other values, including expanded objects and scalars. These are not references and so referential integrity is not an issue.
- Object oriented programs use sub-typing relationships extensively, including the case where the actual result type of a method call is a sub-type of the methods declared return type. The entitlements mechanism must fully support this sub-typing for results in entitlement form.
- Furthermore, there is an additional requirement to support sub-typing in relation to the mechanism for making objects available over the network (implementation of the *REMOTE* interface). The requirement is that such objects may be assigned to variables of any type to which they conform, including types that do not implement the *REMOTE* interface. This may

complicate the design because, as described above, objects that implement the remote interface behave differently in relation to referential integrity when compared to those that do not implement the *REMOTE* interface.

7.2.3.4 Type Orthogonality

In order to support automatic synchronisation, it is necessary to divide results into two categories determined by the type of those results. The categories are:

- The result is an object of a type that is accessed solely by application of polymorphic method calls. Note that this includes all references to objects that implement the *REMOTE* interface.
- The result is an object of a type that is accessed directly in some or all cases, either by access to its instance variables or by the application of non-polymorphic (statically bound) method calls.

This chapter will use the terms *purely indirect access* and *partially direct access* to describe these cases. There are significant differences between these two categories in terms of synchronisation requirements:

- In the former case, it is relatively straightforward to support lazy synchronisation. This is achieved by replacing the original definition of each polymorphic method with one that performs synchronisation and then calls the original definition. This is exactly the approach pioneered by Caromel [Caromel 1993], and there is no reason not to adopt a version of it for entitlements.
- In the latter case, the result may be accessed directly, so it is not possible to take advantage of polymorphic dispatch in the same way. There are two options:
    - Support lazy synchronisation by using the computer's memory protection hardware to detect the first access to the result, delivering a page fault on that first access.
    - Synchronise more eagerly, ensuring that the result is available before it becomes subject to direct access.

In light of these differences, the aim must be to develop a collection of entitlements mechanisms (one for each case above) rather than a single mechanism. This collection of mechanisms should exhibit a uniform interface that is orthogonal to the type of the result.

7.2.3.5 Contract Support

The entitlements mechanism should support contract based programming. A contract for a synchronous local method call involves the following static and dynamic semantics:

1. Type-checking of the arguments to the call.
2. Transmission of the actual parameters to the method.
3. Type checking of the result.
4. Verification of the (assertion) pre-conditions of the call, including the invariant of the object of the call.
5. Verification of the post-conditions of the call.
6. If an exception is raised in the execution of the call, propagation of that exception to the caller.
7. In the absence of an exception, transmission of the result of the call, making it available to the caller.

A synchronous remote method call is similar, except that steps 4 and 5 occur on the remote node, and may involve delay. A deferred synchronous remote method call, such as that supported by entitlements, splits the call submission phase (steps 1 to 3) from the call return phase (steps 6 and 7). The former continues to occur at the point of call, but the latter occurs when the entitlement is used after fulfilment. The entitlements mechanism must therefore support both normal and abnormal call return in order to support contracting.

7.3 The Design of a Mechanism Supporting Entitlements

The description of the design is divided into the two main cases identified in the discussion of the type orthogonality goal, these being:

- Entitlements to references to objects accessed only through polymorphic method calls.
- Entitlements to other values.

As mentioned previously, each of these is supported by a distinct mechanism, with the two mechanisms attempting to present a uniform interface. In each case, entitlement values are created by operations on local objects called proxies that represent remote objects. Proxies are briefly discussed in Section 7.3.1. The remaining subsections describe the three entitlements mechanisms.

7.3.1 Proxies

Suppose one has a class exporting two methods as follows:

```
class SERVER conforms REMOTE feature
  m1(p1 : INTEGER): LIST is …
  m2(p1 : LIST) is …
end
```

The class list does not conform to the *REMOTE* interface, and therefore it is passed by value (closure copy). Consequently, references to *SERVER* objects are local references, which is unlikely to be satisfactory in a distributed system!

References to remote objects of this class are represented in client address spaces by objects of a proxy class that conforms to the type of the *SERVER* class as well as to the *REMOTE* interface, as follows:

```
class SERVER-PROXY conforms SERVER,REMOTE feature
  m1(p1:INTEGER): LIST is …
  m2(p1:LIST) is …
end
```

The conformance relationship between the two classes means that values of the proxy class can be attached to variables of the type of the *SERVER* class. Consequently, a client could achieve uniform access to a group of *SERVER* objects, some of which were local and accessed directly and some were remote and accessed via proxies. This support for sub-typing meets one of the design goals.

The implementation of each method of a proxy involves network operations to locate and call the remote object that the proxy represents. Proxy classes are not normally written by programmers but rather generated (by a stub compiler) from the class of the remote object that the proxy represents.

Design of proxies to support synchronous remote method calls is well understood and examples of implementations are widely available [Sun 1997a]. To support entitlements, with associated asynchrony, a design for proxies is required where proxies continue to be sub-types of the remote object type but with results of remote calls being represented by entitlements rather than by actual values.

7.3.2 Entitlements to Indirectly Accessed Results

Several events can be identified in the life cycle of an entitlement to a result. These events are as follows:

- Creation of the entitlement.
- Supply of the result value for the entitlement.

```
m1(p1 : INTEGER): ENT[LIST] is
  local
    -- Allocate context object for entitlement.
    ctxt : PIA-ENTCTXT := new PIA-ENTCTXT;
    -- Create entitlement object, associate with context.
    ent : ENT[LIST] := ENT[LIST].create(ctxt);
  do
    -- Marshal and send request, including ID of context object.
    ...
    -- Record mapping from ctxt to ent.
    ent.get-ctxt.add-to-entset(ent);
    -- Return new entitlement object.
    Result := ent;
  end
```

Figure 55 – Example Proxy Method Returning an Entitlement.

- Fulfilment of the entitlement upon arrival of the result value.
- Operations on the entitlement, both before and after fulfilment, and including deallocation of the entitlement.
- Transmission of the entitlement in remote method calls and other messages.
- Exception handling.

Each of these events is described in a subsection below.

7.3.2.1 Creation

As mentioned previously, entitlements are created by calls of methods on proxies representing remote objects. Recall the example remote object class:

```
class SERVER conforms REMOTE feature
  m1(p1 : INTEGER): LIST is ...
  m2(p1 : LIST) is ...
end
```

Given that *LIST* is an indirectly accessed object class, the proxy class for the *SERVER* class is as follows:

```
class SERVER-PROXY conforms SERVER,REMOTE feature
  m1(p1:INTEGER): ENT[LIST] is ...
  m2(p1:LIST) is ...
end
```

In which *ENT[LIST]*[97] is a sub-class of *LIST*. It is here assumed that the language supports covariance of method results, whereby the result type of a given method in a sub-type may be a sub-type of the result type of the corresponding method in the super-type. The assumption of this support is useful, but not actually essential – the technique will still work satisfactorily in languages such as Java that do not permit result covariance.

Entitlements are created by calls to proxy methods such as *m1*, which is defined along the lines shown in Figure 55. Note that this technique will work even if result covariance is not supported, in which case the return type is required to be *LIST*.

The code of the class *ENT[LIST]* is generated by the entitlements mechanism (in this it is similar to the proxy class). The code of this class, is shown in part in Figure 56. All entitlement classes in the

[97] The *ENT[LIST]* class is a normal (non-generic) class occupying part of the namespace defined by the *ENT* generic class, for reasons that will become apparent later. The idea of providing a class as the definition of a particular instance of a generic class is derived from the template facilities of C++ [Stroustrup 1991].

```
class ENT[LIST]
  inherit PIA-ENT-BASE,
  conforms LIST
feature {NONE}
  -- Data structures.
  ...
  -- Private constructor, just calls constructor of ENT-BASE.
  constructor(ctxt : PIA-ENTCTXT) : PIA-ENT-BASE(ctxt) is do end
feature {ENTCORE}
  -- Creation operation.
  create(ctxt : PIA-ENTCTXT) : ENT[LIST] is
     do
        Result := new ENT[LIST] (ctxt);
     end
feature
  -- Publicly accessible methods.
end
```

Figure 56 – Example of an Entitlement Class for a Purely Indirectly Accessed Class.

```
class PIA-ENT-BASE
feature {NONE}
  ctxt : PIA-ENTCTXT;
feature {PIA-ENT-BASE*}
  -- Constructor accessible only to sub-classes.
  -- Simply initialises ctxt instance variable.
  constructor(ctx : PIA-ENTCTXT) : ctxt(ctx) is do end
feature
  -- Publicly accessible operations
end
```

Figure 57 – PIA_ENT_BASE Class.

purely indirectly accessed category inherit from the *PIA_ENT_BASE* class. This is shown in Figure 57. These class definitions will be augmented as the design is elaborated below.

7.3.2.2 Supply

Recall that the methods of a server proxy class responsible for creating entitlements send a request message to the server object represented by that proxy. Also, recall that this request message includes a copy of the identity of the context (*PIA_ENTCTXT*) object of the entitlement. The arrival of the request message at the server triggers the execution of the requested method, as it does with conventional remote method invocation.

The method execution eventually completes, returning a result value. This result value is marshalled, and it, together with the identity of the context object, is sent back to the primary demander for the entitlement in a reply message. This process implements the supply operation – the responsible supplier for the entitlement has fulfilled its responsibility.

It may be, however, that the value supplied to fulfil the entitlement is later required on the responsible supplier as well as on the primary demander. For this reason, the marshalled value of the result is inserted into a lookup table indexed by a key resulting from the concatenation of the network identity of the primary demander and the identity of the context object. To ensure uniqueness of the key, it is necessary both that network identities be unique across the system and that entitlement context identities be unique within a given network identity. The values stored in the table will be used if the entitlement value is later received in method calls to the responsible supplier.

```
class ENTCORE
feature
  fulfil-and-resume(ctxtid : PIA-ENTCTXT-ID, res : ANY) is
     local
        -- Find the context object.
        ctxt : PIA-ENTCTXT := lookup-ctxt(ctxtid);
     do
        foreach ent in ctxt.entset() loop
           -- Fulfil each copy of the entitlment with the result of
           -- applying the clone() method to res. If res is a proxy
           -- for a REMOTE object, this will simply return the proxy.
           -- Otherwise, clone() will return a copy of the closure of
           -- res.
           ent.ent-fulfil(res.clone());
        end
        -- Resume all threads waiting on the entitlement.
        ctxt.get-waiting-threads().resume-all();
     end
  -- Other methods
end
```

Figure 58 – ENTCORE.fulfil_and_resume Method.

7.3.2.3 Fulfilment

The responsible supplier for an entitlement will send a reply message to the primary demander, as described in Section 7.3.2.2. When this message is received and unmarshalled, the process of fulfilment of the entitlement begins.

The context identity and the unmarshalled value of the result are passed to the *ENTCORE. fulfil_and_resume*() method, which is shown in Figure 58. This method locates the context object and then uses the unmarshalled result value to fulfil the entitlement. For reasons that are detailed later, transmission of entitlements to non-*REMOTE* reference objects in remote method calls can lead to there being several entitlements associated with a given context object, although there will usually be only one. Each of these copies of the entitlement represents a distinct object, consequently each must be fulfilled individually, with a deep-copy of the result. After all the copies are fulfilled, any threads waiting to perform operations on the results represented by any of these copies are resumed.

The *ent_fulfil*() method is defined in the *PIA-ENT-BASE* class. There are several ways to implement this method, with the choice determined by the language environment. Two general approaches are:

- ***Forwarding*** – Assign the object passed to *ent_fulfil*() to an instance variable in the entitlement object. Implement all operations on the entitlement object in such a way that they are invisibly forwarded to apply to the object attached to this field, rather than to the entitlement object.
- ***Reference Redirection*** – In some fashion, alter all references that (before fulfilment) pointed to the entitlement object so that (after fulfilment) they point to the result object.

These two alternatives impact on the mechanism for user and system defined operations on entitlements; this is discussed later. The present discussion focuses solely on the fulfilment operation.

7.3.2.3.1 Forwarding

If forwarding is used, then the *PIA_ENT_BASE* class simply declares the *ent_fulfil*() method and a supporting method called *ent_valid_fulfilling_type*() without providing implementations. An additional data structure is added to the entitlement class, which also defines these methods. The

```
class ENT[LIST]
  inherit PIA-ENT-BASE,
  conforms LIST
feature {NONE}
  -- Data structures.
  res : LIST;
feature {ENTCORE}
  ent-valid-fulfilling-type(raw:ANY) : BOOLEAN is
     do
        -- Return whether raw can safely be cast to type LIST.
        Result := (dynamic-cast[LIST](raw) /= null);
     end
  ent-fulfil(raw : ANY) is
     require
        ent-valid-fulfilling-type(raw)
     do
        -- Cast that replicates precondition, hence always succeeds
        res := dynamic-cast[LIST](raw);
        -- Mark entitlement as being fulfilled.
        ctxt := null;
     end
  -- Other features given elsewhere.
end
```

Figure 59 – Example of Forwarding Fulfilment

ENT[LIST] example is shown in Figure 59. The code simply checks the type of the raw result, which should always conform to *LIST*, and assigns it to the instance variable.

The main advantages of this approach are that it is simple, that it works in all object-oriented languages and that it does not make unusual requirements of the memory management sub-system. The main disadvantage is that it interposes an extra level of indirection in all method calls on result objects, leading to a loss of performance. It may also be wasteful of memory, because the original entitlement object cannot be reclaimed once the result value is available.

7.3.2.3.2 Reference Redirection

If reference redirection is used, then *ent_fulfil*() is implemented in the *PIA_ENT_BASE* class, as shown in Figure 60. It is assumed that there is a system level operation *MEMORY.redirect*(*p1*,*p2*) which takes two pointers and causes all references to the second to refer to the object referred to by the first.

The implementation of *MEMORY.redirect*() may be quite challenging. Two general approaches can be adopted:

- Scan through all pointers in memory (and in registers), changing all pointers containing the value *p2* to have the value *p1*. This obviously requires the ability to distinguish pointer values from scalars having the same value at the representation level. Furthermore, in languages supporting multiple implementation inheritance there may be pointers to sub-objects at a non-zero offset within the representation of a given object, the redirect operation must account for this possibility, leading to a significant increase in complexity.
- Use a double indirection scheme for all references. In this scheme, object references, such as *p2*, refer to master pointer objects, these then refer to the actual object referenced by the pointer. Changing all copies of *p2* to point to some other object is simply a matter of updating the value in the master pointer. Many Java implementations use double indirection in object references.

```
class PIA-ENT-BASE
  -- Other features given elsewhere.
feature {ENTCORE}
  ent-fulfil(raw : ANY) is
     do
        -- Redirect all pointers to this object to point to raw.
        MEMORY.redirect(raw,this);
     end
end
```

Figure 60 – Supporting Reference Redirection with the PIA_ENT_BASE Class

```
class PIA-ENT-BASE
  -- Other features given elsewhere.
feature {NONE}
  ent-wait-imp is
     local
        current : THREAD := THREAD.current();
     do
        ctxt.get-waiting-threads().add(current)
        current->suspend()
     end
feature
  ent-waiting: BOOLEAN is do Result := (ctxt /= null) end
  ent-fulfilled: BOOLEAN is do Result := (not ent-waiting) end
  ent-wait is
     do
        if ent-waiting then
           ent-wait-imp
        end
     end
end
```

Figure 61 – Explicit Synchronisation Operations in the PIA_ENT_BASE Class.

The main advantages of this approach are that method calls to results are just as fast as method calls to purely local objects and that the memory of an entitlement object may be reclaimed once the result value is available. The main disadvantages are that it may have a unpredictable impact on performance and that it imposes special requirements on the memory management sub-system. Finally, this approach cannot be implemented at all for languages (such as C++) which do not use double indirection and in which pointers cannot be identified with complete accuracy.

7.3.2.3.3 The Perils of Overwriting

One might attempt to use another approach in implementing fulfilment. The idea is to implement fulfilment by overwriting the memory of the entitlement with a copy of the result (note that this memory includes any infrastructure used to implement polymorphic method calls). This would have the advantage that any references to the entitlement object will automatically refer to the result after fulfilment. However, this will not always work since the actual value of the result may be a sub-type of the result's declared type and may therefore be arbitrarily large. One could, of course, implement a hybrid of this overwriting scheme combined with pure forwarding, using overwriting where it is possible and pure forwarding where it is not, but this seems a great complication for little gain.

7.3.2.4 Operations on Entitlements

With one exception, most categories of operations on an entitlement conceptually apply to the result represented by the entitlement, rather than to the entitlement itself. The categories of operations are:

```
ent-waiting: BOOLEAN is
   local
      -- Assign pib a non-null value if this is a sub-class
      -- of PIA-ENT-BASE.
      pib : PIA-ENT-BASE:= dynamic-cast[PIA-ENT-BASE](this);
   do
      -- If the entlement is unfulfilled, then it will be an object
      -- of a sub-class of PIA-ENT-BASE and pib will be non null.
      Result := (pib /= null)
   end
```

Figure 62 – Explicit Synchronisation (ent_waiting method) Revised for Reference Redirection.

- Explicit synchronisation operations. These are defined by the entitlements mechanism. This is the only category of operations that applies to an entitlement rather than to the result it represents.
- User-defined operations.
- Comparison of the result for reference equality.
- Dereference of the result to obtain the value of the attached object.
- Operations using the actual (dynamic) type of the result.

7.3.2.4.1 Explicit Synchronisation Operations

Entitlements are primarily intended to be used with implicit synchronisation enforced when operations are applied to results. Sometimes, however, explicit synchronisation may be desirable, so the *PIA_ENT_BASE* class defines several explicit synchronisation operations, as shown in Figure 61. In order to use these operations, it will be necessary for the client to have a reference to the entitlement that are declared to be of the entitlement type, rather than of the declared result type. This will require modification of the client code where explicit synchronisation is needed, but modification is needed in any event to add the explicit synchronisation calls, so this is not a major issue.

A complication arises if the reference redirection approach to fulfilment is used. Prior to fulfilment, there may be references to the entitlement object and these may be of the entitlement type (*ENT[LIST]* in the example), which is a sub-type of *PIA_ENT_BASE*. These references point to an object of a sub-class of *PIA_ENT_BASE*. After fulfilment, these references point to a different object, the result value. The result is not necessarily an object of a sub-class of *PIA_ENT_BASE*. Consequently, the referenced object may not have a *ctxt* instance variable nor does this object necessarily define the explicit synchronisation methods (or it may redefine one or more of them). This complication is strictly a type error that arises because the reference redirection operation operates below the type system level. However, it is possible to work around the problem and avoid having to discard the reference redirection approach, as follows:

- All the explicit synchronisation methods should have static binding, meaning that calls to them are direct, rather than indirected via the mechanism for dynamic binding of polymorphic method calls. This ensures that there is always a method to be called and that it is the implementation defined in *PIA_ENT_BASE* that is executed.
- The implementation of the *ent-waiting* method should include a dynamic type check to determine whether the object to which it is applied is an entitlement object. In fact, as shown in Figure 62, the result of the dynamic type check is sufficient to determine the result of the call, obviating the need to access the *ctxt* variable.

A further anomaly remains. Consider a case in which the result supplied to fulfil an entitlement is itself an entitlement value. This might be extended so that the fulfilling value of the entitlement supplied as the result for the original is itself an entitlement, and so on. In these cases, the implementations of explicit synchronisation given above will exhibit different behaviour, as follows:

- The forwarding implementation will indicate that the original entitlement has been fulfilled. It does not provide any way to determine fulfilment of the entitlement returned as result, or of any of the subsequent entitlements in the extended case.
- The reference redirection implementation will return information pertaining to the entitlement supplied as the result, but there is no way for the user to determine that the synchronisation information relates to this entitlement, rather than to the original. In the extended case, the synchronisation information will relate to the most recently delivered result.

These alternatives in the semantics are assessed as follows:

- The caller of the original request submitted that request submitted that call in order to obtain a (non-entitlement) result value. The second semantics provides the caller with information about the availability of this result value. This information is independent of any process by which the result value is provided (whether by a single entitlement or a series of entitlements). In most cases, this is advantageous, since the caller is not interested in the process, only in the result. In contrast, the first proposed semantics gives the caller information about the first stage of the process of result provision, a detail that is likely to be superfluous. Furthermore, this semantics only reports information about the first stage, so with a multi-stage result provision, the caller cannot obtain the information that is most often actually required, namely whether the result value is available.
- The framework presented in Chapter Five of this thesis introduced the notion of coordinated termination. The idea is that one node involved in an interaction had sufficient information available to it to determine that all communication and computational activity associated with the interaction had completed. This node is termed the termination coordinator. In typical scenarios, a client submitting requests is the termination coordinator for an interaction involving those requests.

  Suppose that a client sends a request to a server, but that instead of supplying a result value for that request, that server delegates responsibility for that request to a second server. It does this by sending a request to that second server and sending back an entitlement to the result of that second request as the result supplied to the client. Consequently, the interaction involves three nodes, the client and two servers.

  The client, acting as termination coordinator, needs to establish that the interaction is complete. The second proposed semantics unblock the client only when both the reply from the first server arrives and the result value arrives after being sent in the reply from the second server[98]. This is exactly the information needed for coordinated termination in this case. In contrast, the first proposed semantics provides information only about the first server.

Consequently, the second alternative should be preferred, and the implementation of the *ent-waiting* method should be revised in the forwarding implementation, as shown in Figure 63. This implementation replicates the behaviour of the reference redirection implementation in the extended case as well as the simple case, in that it will follow the chain of *res* pointers to the most recently received result.

[98] There might appear to be a complication when the reply from the second server arrives first. However, this is dealt with by the general mechanism for unmarshalling messages containing entitlements received by nodes other than the primary demander and responsible supplier, which is discussed in Section 7.3.2.5.4.

```
ent-waiting: BOOLEAN is
   local
      -- Assign pib a non-null value if res is a sub-class
      -- of PIA-ENT-BASE.
      pib : PIA-ENT-BASE:= dynamic-cast[PIA-ENT-BASE](res)
   do
      -- If the entlement has been fulfilled with an unfulfilled
      -- entitlement, then res will be an object of a sub-class of
      -- PIA-ENT-BASE and pib will be non null. In this case, the
      -- synchronisation test should be forwarded to the object
      -- attached to res, otherwise it should be evaulated as
      -- before.
      if pib /= null then
         Result := pib.ent-waiting
      else
         Result := (ctxt /= null)
      end
   end
```

Figure 63 – Explicit Synchronisation (ent_waiting method) Revised for Forwarding.

7.3.2.4.2 User-Defined Operations – the Forwarding Approach

The *ENT*[*LIST*] example is extended in Figure 64 to show the pure forwarding implementation of a user defined operation on the result object. The pure forwarding approach requires an extra class, *ENT_SYNC*[*LIST*] in the example, to be generated for every class of result subject purely to indirect access.

As shown in Figure 64, the implementation of each user-defined operation is spread across the two generated classes (*ENT*[*LIST*] and *ENT_SYNC*[*LIST*] in the example). The *ENT*[*LIST*] version of each operation simply calls the corresponding method of the object attached to the *res* instance variable, passing any parameters and returning the result (if any). After fulfilment, as described earlier, the *res* instance variable will point to the object supplied as the result of the call, so any call will immediately be forwarded to the result object.

Prior to fulfilment, the *res* field points to an object of type *ENT_SYNC*[*LIST*], this is created as part of the construction of the *ENT*[*LIST*] object. The *ENT_SYNC*[*LIST*] object contains a reference (*back_ref*) back to the *ENT*[*LIST*] object. The implementation of each user defined method in the *ENT_SYNC*[*LIST*] class blocks, waiting for the entitlement to be fulfilled. The blocking is implemented by calling the *ent_wait* explicit synchronisation method defined in *PIA_ENT_BASE*. It then calls the corresponding method of the object attached to the *res* variable of the object attached to *back_ref*. This is also the result object, since fulfilment updates the entitlement's *res* variable. This process is summarised in Figure 65.

7.3.2.4.3 User-Defined Operations – the Reference Redirection Approach

An example of the reference redirection implementation of user-defined operations is shown in Figure 66. Each method implementation simply waits for fulfilment then calls the method again, relying on the fact that reference redirection has changed the target of the method call. To counter possible special treatment of the pointer to **this** by the memory system, this second call is made through an auxiliary pointer initialised to reference the object referenced by **this**. This auxiliary pointer will be found and updated by the *redirect* operation.

7.3.2.4.4 Comparison of the Result

Comparison has straightforward semantics if two entitlements are compared or if two non-entitlement reference values are compared. In each case comparison is defined as pointer comparison. In contrast,

```
class ENT[LIST]
  inherit PIA-ENT-BASE,
  conforms LIST
feature { ENT-SYNC] }
  -- Data structures.
  res : LIST;
  -- Private constructor, calls constructor of ENT-BASE.
  constructor(ctxt : PIA-ENTCTXT) : PIA-ENT-BASE(ctxt) is
     do
        res := new ENT-SYNC[LIST] (this)
     end
  -- Other features given elsewhere.
feature
  -- User defined methods
  lm1(param:INTEGER) : BOOLEAN is
     do
        Result := res.lm1(param)
     end
end

class ENT-SYNC[LIST]
  conforms LIST
feature { ENT[LIST] }
  -- Data structures.
  back-ref : ENT[LIST];
  -- Private constructor
  constructor(ent : ENT[LIST]) is do back-ref := ent; end
  -- User defined methods
  lm1(param:INTEGER) : BOOLEAN is
     do
        back-ref.ent-wait
        Result := back-ref.res.lm1(param)
     end
end
```

Figure 64 – Forwarding Implementation of User-defined Operations.

when one of the values to be compared is an entitlement and the other is not, complications arise. In particular:

- If a fulfilled entitlement is compared to a non-entitlement value, then the comparison should apply to the fulfilling result value of the entitlement rather than to the entitlement.
- If an unfulfilled entitlement is compared to a non-entitlement value then the comparison should block until the entitlement is fulfilled.

Note that the first case cannot arise with reference redirection since references to entitlements are replaced by references to the fulfilling value at the time of fulfilment.

Supporting each of the cases identified above requires redefinition of the semantics of comparison when one of the two operands is an entitlement value. Furthermore, due to polymorphism, the actual types of the operands are not known statically, so the determination of the comparison semantics that apply must be done dynamically. Unfortunately, the standard mechanism for polymorphic method call cannot be used to provide the appropriate semantics. The problem is that this mechanism dispatches based on the actual type of only one of the operands to any operation (the first operand for a dyadic operator and the call target, *o*, in a method call). Consequently, this approach provides a redefined semantics only for comparisons where the actual type of the left hand side operand is an entitlement value. It fails for comparisons where the right hand side operand is an entitlement and the left hand

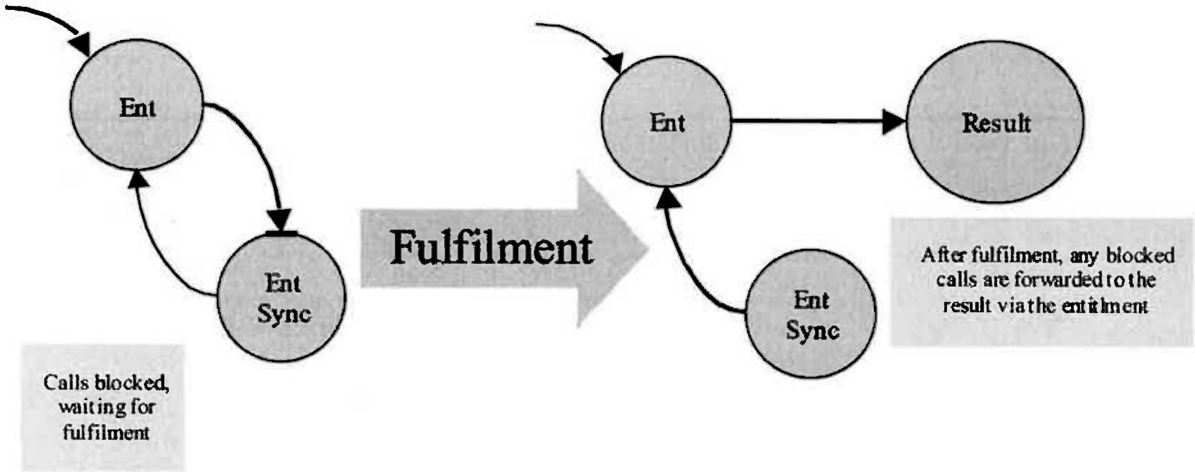

Figure 65 – Call Forwarding Illustrated.

```
class ENT[LIST] inherit PIA-ENT-BASE, conforms LIST
feature {NONE}
  -- Private constructor, just calls constructor of ENT-BASE.
  constructor(ctxt : PIA-ENTCTXT) : PIA-ENT-BASE(ctxt) is do end
  -- Other features given elsewhere.
feature
  -- User defined methods
  lm1(param:INTEGER) : BOOLEAN is
     local
        -- Obtain a reference to the current object
        me : LIST := this;
     do
        ent-wait
        -- The me reference has been updated by fulfilment, so
        -- method calls through me will now apply to the result.
        Result := me.lm1(param)
     end
end
```

Figure 66 – Reference Redirection Implementation of User-defined Operations.

side is not. . Such a mechanism would exhibit very strange semantics indeed! Furthermore, since the actual type information is not known at compile time, it is not feasible to reverse the order of the operands, even though such a reordering is permitted in principle by the symmetry of equality.

The multi-method [Bobrow *et.al.* 1986, Bobrow *et.al.* 1988, Millstein and Chambers 1999] approach to polymorphic dispatch allows the operation that is called to be determined by the actual types of several of the operands and would be suitable for this purpose. However, due to the lack of efficient implementation techniques, multi-methods are not widely implemented. Consequently, the entitlements system is forced to require alterations to existing user classes (the *ENT[LIST]* class in the example) in order to achieve the desired semantics for comparison.

7.3.2.4.5 Dereference of the Result

To maintain consistency with the semantics of the other operations, an attempt to dereference an entitlement should block until that entitlement is fulfilled and then should return the result of dereferencing the result fulfilling the entitlement. In contrast to comparison, dereference is a monadic operation with the entitlement as its single operand. Therefore, the conventional polymorphic dispatch mechanism can be used to achieve the desired semantics, assuming that the language permits polymorphic redefinition of the dereference operation.

7.3.2.4.6 Operations on the Actual Type of the Result

Modern object-oriented languages typically support certain operations related to the actual type (sometimes called the dynamic type) of a value. For example, there might be an operation to determine whether a given value conforms to a given type (this is essentially the purpose of the ***instanceof*** operator in Java).

When applied to an entitlement to a result the application of these operations should be forwarded to apply to the actual result represented by the entitlement, rather than the entitlement itself. It is this actual result that is meaningful in the program using the entitlements mechanism, and consequently the programmer might very well have a legitimate interest in this result's actual type. In contrast, the entitlement object is an artefact of the entitlements mechanism. Such being the case, the programmer has no legitimate interest in the exact type of this object (recall that there is a separate mechanism to discover whether a result is currently in an unfulfilled or fulfilled state).

7.3.2.5 Transmission of Entitlements

Consider the following code:

```
s : SERVER
r : LIST := s.m1(56)
s.m2(r)
```

The value attached to the variable *r* is an unfulfilled entitlement until it is fulfilled by the return of the result. In particular, it may still be an unfulfilled entitlement value when the call on the last line of the code fragment is marshalled.

This raises the question of how to handle attempts to send unfulfilled entitlements in messages. A fundamental requirement for entitlements is to avoid blocking in these situations as much as is feasible. There are three cases as follows:

- The case where an unfulfilled entitlement is sent back to the node containing the server for the remote method invocation responsible for fulfilling the entitlement. That is, the entitlement is sent to its responsible supplier node. This is the case shown above.
- The case where an unfulfilled entitlement, having been sent to some other node or nodes, is sent back to the primary demander node for that entitlement.
- The case where an unfulfilled entitlement is sent to some node other than the primary demander or responsible supplier.

Somewhat different implementations are required for each of these cases, which are therefore discussed separately in Sections 7.3.2.5.2, 7.3.2.5.3 and 7.3.2.5.4 below. The marshalling process is common to all cases, this is discussed in Section 7.3.2.5.1.

7.3.2.5.1 Marshalling

Unfulfilled entitlement values passed in requests are marshalled into a form including information from the *PIA_ENTCTXT* object associated with the entitlement. This includes the identity of the context (*PIA_ENTCTXT*) object on the primary demander node and the following information:

- The network identity of the responsible supplier node for the entitlement.
- The network identity of the primary demander node for the entitlement.

The unmarshalling process compares this information with the identity of the local node. Using this comparison, it is able to determine which of the three case described in the sections below should apply for a given unfulfilled entitlement value received in a network message. As discussed previously, the context identity and identity of the primary demander node together uniquely identify entitlements across the system. The concatenation of these two identities is termed the *entitlement key*. The entitlement key is also sent with the original request associated with the entitlement.

7.3.2.5.2 Messages Received by the Responsible Supplier Node

When an unfulfilled entitlement is encountered by the unmarshalling process on that entitlement's responsible supplier node, the system performs a lookup in a table of produced results. This lookup uses the entitlement key of the entitlement to locate entries in the lookup table.

If the lookup succeeds, then the method call associated with the entitlement has completed and produced a result. The lookup returns a copy of that result. The unmarshalling process substitutes a copy of this object and its closure into the unmarshalled message. Second and subsequent references to the entitlement in that message are substituted with references to this copy. However, each message containing references to the entitlement is provided with a private copy of the closure, in order to preserve the semantics of call by value.

If the lookup fails then it indicates that the method responsible for supplying the result fulfilling the entitlement has not completed execution. In this case, the unmarshalling process is suspended until the method completes. This case occurs rarely, since requests from a given client (node) are serviced in the order sent, and unmarshalling of a given such request is not commenced until its predecessor has completed execution. The most probable cause is that the message containing the unfulfilled entitlement is a request or reply sent by a third party. Suspension and resumption of the marshalling process handles that case effectively. A more pathological case occurs when the message containing the original request from the client is lost, a timeout based failure detector handles this (extremely rare) situation.

Once the unmarshalling process completes, the unfulfilled entitlement values have been replaced by their fulfilling values, and the message can be processed as it would normally.

In order to support this unmarshalling process, the skeleton code that sends the result of a method call back to the client is augmented so that it installs that result into the table of pending results. This process also uses the entitlement key to place the result into the table.

Suspension of the unmarshalling process for requests does not introduce significant additional susceptibility to deadlock. Let *UI* be an unfulfilled entitlement. Let *m1* be the method call responsible for producing the result fulfilling *UI*. Let *m2* be the method call that is being unmarshalled. Now, *m2* depends on the result of *m1*. Deadlock involving the suspension due to unmarshalling can only occur if the execution of *m1* depends on the execution of *m2* in some way. The dependency cannot be on the result of *m2*, since the fact that *m2* is dependent on the result of *m1* indicates that *m2* was issued causally later than m1, although by a different node[99]. This causal relationship means that the entitlement representing the result of *m2* was not in existence when *m1* was issued. In any case, a set of method calls cyclically dependent for their execution on the results of other method calls in the set would obviously be a programming error. Consequently, the dependency must be on some part of the execution of *m2* that is not dependent on the result of *m1* passed in the request triggering *m2*. It is possible to construct examples where such a dependency exists. However, the probability of this case occurring in practice appears to be vanishingly small.

Suspension of the unmarshalling process for replies can deadlock in the presence of cyclically dependent entitlements. This situation can be detected and dealt with, as described in Section 7.3.2.5.5

[99] Recall that suspension of unmarshalling cannot occur when both requests are from the same node, unless there has been a message loss, in which case a timeout-based failure detector will terminate the suspension.

Garbage collection of the pending results table is a problem because unfulfilled entitlements can be sent to arbitrary nodes. The garbage collection problem is therefore global to the system, and a general distributed garbage collection mechanism is required[100].

An important consequence of this unmarshalling process is that a node never contains unfulfilled entitlement references for entitlements for which that node is the responsible supplier. Consequently, it is impossible for an unfulfilled entitlement to be supplied as the fulfilling value for that same entitlement (nor as part of that fulfilling value). However, cycles involving more than one entitlement can occur; this is discussed in Section 7.3.2.5.5.

7.3.2.5.3 Messages Received by the Primary Demander Node

When an unfulfilled entitlement is encountered by the unmarshalling process on that entitlement's responsible supplier node, the system creates a copy of the entitlement object (an object of class *ENT[LIST]* in the *LIST* example). This is associated with the existing *PIA_ENTCTXT* object for the unfulfilled entitlement by adding the copy to set of entitlements associated with the *PIA_ENTCTXT*. The unfulfilled entitlement is replaced by the copy in the unmarshalled request. Second and subsequent references to a given unfulfilled entitlement encountered in the unmarshalling of a given request are replaced by references to the copy of the entitlement created when the first reference in that request was encountered. Consequently, each unmarshalled request has a single private copy of the entitlement. This is necessary to maintain the correct semantics for objects passed by value.

As described previously, when the reply containing the result fulfilling the entitlement arrives, the *ENTCORE.fulfil_and_resume*() routine shown in Figure 58 (see page 201) is called by the system. The need for the loop in this routine is now apparent: each iteration of the loop fulfils a copy of the entitlement object, and later iterations create a copy of the closure of the result.

It is possible for programs to clone an unfulfilled entitlement object. The semantics of cloning the entitlement after it is fulfilled are to create an independent copy of the closure of the result fulfilling the entitlement. This semantics is supported by creating a copy of the unfulfilled entitlement in the same way as was done when an unfulfilled entitlement was unmarshalled. The *ENTCORE.fulfil_and_resume*() routine then updates that copy as well.

7.3.2.5.4 Messages Received by Other Nodes

When an unfulfilled entitlement is encountered by the unmarshalling process on a third party node, three cases are identified:

1. The case where the entitlement has not previously been received by the node..
2. The case where the entitlement has previously been seen by the node but is encountered for the first time in the unmarshalling of a given request.
3. The case where the entitlement is encountered for the second or subsequent time in the unmarshalling of a given request.

In the first case, the unfulfilled entitlement is unmarshalled into an object of the entitlement class (*ENT[LIST]* in the *LIST* example). A new *PIA_ENTCTXT* object[101] is created and associated with this entitlement object. Importantly, the state of this *PIA_ENTCTXT* object includes a copy of the identity of the *PIA_ENTCTXT* object on the primary demander node for the entitlement. The *PIA_ENTCTXT* object on the primary demander is termed the master *PIA_ENTCTXT* object. The marshalling process described in Section 7.3.2.5.1 uses the identity of this master *PIA_ENTCTXT*, rather than the address

[100] Such a mechanism is required in any event to support garbage collection of distributed graphs of ordinary objects, so this is not really an additional requirement.

[101] Importantly, the state of this *PIA_ENTCTXT* object includes a copy of the identity of the *PIA_ENTCTXT* object on the primary demander node for the entitlement. The *PIA_ENTCTXT* object on the primary demander is termed the master *PIA_ETCTXT* for the entitlement. The marshalling process described in Section 7.3.2.5.1 uses the identity of this master *PIA_ENTCTXT*, rather than the address of the non-master *ENTCTXT* created on a third party node.

of the non-master *PIA_ENTCTXT* created on a third party node. The entitlement object is substituted into the unmarshalled form of the request. In addition, a *demand registration* message is sent to the responsible supplier node for the entitlement requesting that that node send a *result propagation* message containing a copy of the result when it is available. The arrival of this result triggers a call to a slightly modified version of the *ENTCORE.fulfil_and_resume* routine in the same way as a reply triggers the *ENTCORE.fulfil_and_resume* routine on the primary demander node. The modification concerns the need to map from the address of the master *PIA_ENTCTXT* object to the local non-master *PIA_ENTCTXT*, the routine is otherwise the same.

To support transmission of the fulfilling value to dependent third parties when it becomes available, the responsible supplier maintains a set of dependent third party nodes for each entitlement it is responsible for. This is initially empty, and a node is added when a dependency registration message from that node is received by the responsible supplier.

As an optimisation, if an unfulfilled entitlement is sent from its responsible supplier to a third party node the responsible supplier adds the destination to the set of dependent third party nodes for the entitlement. Consequently, the third party will be sent the fulfilling value of the entitlement as soon as it becomes available, without having to request it via a demand registration message[102].

In the second case, the system locates the local non-master *PIA_ENTCTXT* object and creates an entitlement object associated with it. This entitlement object is then substituted into the unmarshalled form of the request.

For the third case, second and subsequent references to a given unfulfilled entitlement in a given request are replaced by reference to the entitlement object created for the first reference in the request. This preserves the referential integrity of reference to objects within a given request.

This aspect of the mechanism is somewhat inefficient in wide area systems. Instead of requiring a round trip to service demands from third party nodes it may be better to collect Mentat-like [Grimshaw 1993] dependency graphs and propagate these to the responsible supplier node. That node would then pro-actively send the value fulfilling the entitlement to any third party nodes that had dependencies indicated in the graph associated with the entitlement.

7.3.2.5.5 Cycles

It is possible for cyclic dependencies to arise between multiple entitlements, whereby an entitlement is indirectly its own fulfilling value. Such cases clearly represent an error in programming logic. It is, however, helpful to be able to detect such errors.

The simplest general case in which a cycle can arise involves a client node, *C*, and two server nodes, *S1* and *S2* (which might actually be the same process as *C*). The client executes a method call to *S1*, obtaining an entitlement *E1*. Server *S2* then in some way obtains a reference to the unfulfilled entitlement *E1*, perhaps it is passed in a method call from *C*. Next, the server *S1* calls a method of *S2*, obtaining an entitlement *E2*. Server *S1* then supplies *E2* as the fulfilling value for *E1*. At this stage the client has a reference to *E1*, which contains a reference to *E2* as its fulfilling value. Server *S2* then supplies its reference to *E1* as the fulfilling value for *E2*. The client then attempts to access *E1*, and is immediately redirected to access *E2*. The client then requests *E2* from its responsible supplier (*S2*), (unless the client is *S2*, in which case it already has *E2*). This brings a copy of *E2* into the client's cache, with *E1* as its fulfilling value. The client is redirected (whether via the cache, or directly, if it is *S2*) to *E1*, and, without further network interaction, will loop infinitely, oscillating between *E1* and *E2*. This situation clearly extends to cycles involving more than two entitlements.

The key to managing these cycles is to notice that before the cycles can become a problem for a given client, all of the entitlements in the cycle must be present in memory on the client. Each

[102] It is possible that a given third party node sends a demand registration message anyway. This message is ignored if that third party node is already in the set of dependent nodes.

entitlement is either a locally created entitlement for which the client is the primary demander, or is remotely created entitlement which is in either the cache of remotely supplied entitlements or in the table of locally supplied entitlements (in which case the client is the responsible supplier). In any event, cycle detection can be performed as a purely local operation on the client. This could be done asynchronously, with a thread that "poisoned" any entitlements involved in cycles, leading to an exception being raised in response to any operations on those entitlements. This is somewhat wasteful, in that quite a lot of unnecessary processing may occur in attempting to resolve entitlements before the poisoner thread catches up with them. Consequently, it may be preferable to build cycle detection into the process of resolving an entitlement into its fulfilling value. In this scenario, each thread has a table of entitlements involved in the currently proceeding resolution, and detects cycles when it attempts to resolve an entitlement that is already in the table. This only detects cycles since each entitlement has only one fulfilling value (out-edge in the graph) and therefore it is not possible for directed acyclic graphs containing braches and joins to be encountered in the resolution process. Once a cycle is detected, all the entitlements are poised as described previously.

7.3.2.5.6 Design Choices: Laziness and Strictness

In each of the three main cases for unmarshalling, there are choices to be made between lazy versus strict approaches to entitlement substitution. In the responsible supplier case, the substitution is strict, whereas in the other cases, it is mostly lazy. This section describes the rationale for the choices made.

In the responsible supplier case, the unmarshalling process strictly substitutes the value fulfilling entitlement for occurrences of that entitlement. The unmarshalling process blocks if strict substitution is not immediately possible. The choice of a strict substitution is motived by the expectation that blocking will be extremely infrequent, and that this infrequency does not justify the additional complexity of a lazy substitution.

In the other two cases, the substitution is mostly lazy. This decision has the very real benefit of supporting non-blocking clone operations on local objects. This is important because cloning is not an operation that would normally be expected to block. One aspect in which these cases are strict rather than lazy concerns the update of entitlement objects when a reply arrives. In these case all copies of the entitlement object are updated, including those on which no threads are blocked. This decision could be expensive where there are large numbers of entitlement copies and the result closure is sufficiently large that the closure copy operation becomes expensive. In this situation, it may be better to do the update process lazily for entitlement values on which no thread is blocked at the time when the reply arrives. It might also be better to resume each blocked thread as soon as the entitlement object on which it is blocked has been updated, rather than resume all threads after all entitlements have been updated.

7.3.2.5.7 Reliability Considerations

The entitlements mechanism runs over an unreliable network. This must be taken into consideration in the design of the communication between nodes as described above. In fact, the lack of network reliability is easily dealt with, since most communication operations used internally by the mechanism fall into the following categories:

- Requests that trigger a side effect free computation and the sending of a reply containing the result of that operation. The main example of this is process used by third party nodes to retrieve the value of an entitlement from the responsible supplier. The third party node can use an at-least-once protocol for this category of operation, resending the request until it receives a reply and thereby masking any (non permanent) network failures.
- Messages that can be discarded (by the entitlements mechanism) if they arrive more than once. The most important example of this is the message from responsible supplier to primary

demander carrying the value of the entitlement. This message can also be transmitted using an at-least-once protocol.

The only aspect of the system that cannot mask network failures is the triggering of remote operations. Request messages cannot, in general, be re-transmitted, since the computations they trigger may not be idempotent. Consequently, remote method execution has at-most-once semantics, as is typically the case with RPC and RMI.

Reliability of the nodes themselves is also an issue. In general, if the responsible supplier for a given entitlement crashes, the value of that entitlement will not be available to nodes requesting it in future. Instead, ant nodes that have not already received a copy of the value will report an exception when an attempt is made to use the entitlement. Replication techniques [Budhiraja *et.al.* 1993, Schneider 1993] could be used to improve availability in this case, but this is beyond the scope of this thesis. In contrast, the failure of any other node (including the primary demander) is not a significant problem. The failed node in this case simply ceases to demand the value of the entitlement. Recall that the identity of an entitlement is derived, in part, from the identity of the primary demander that created it. Consequently, the system must ensure that a restarting node must not assume the same identity as a failed node that has created entitlements that are still extant.

7.3.2.6 Exception Handling

Entitlements are fundamentally a mechanism for deferred synchronisation with the arrival of replies, so it is natural that exception handling is built around synchronisation with replies. Exceptions are of two kinds:

- Those that are raised in the execution of the remote method and that are not handled within that execution. These must be propagated back to the caller.
- Those raised by the middleware, to report failures such as that a remote call could not be reliably completed.

Note that the second kind of exception can always be raised in a system with at-most-once semantics, even if the first kind of exception is prohibited by a restriction preventing remotely callable methods from raising user defined exceptions.

As discussed in the framework, the primary design issue concerns how exceptions are made available to the caller for handling. This issue is independent of the particular kinds of exceptions listed above.

With entitlements, by default, exceptions raised as a result of a remote method call are made available at the point(s) at which the client synchronises with the reply associated with the call. This may be an explicit synchronisation operation, or more likely, it will be implicit synchronisation result from a strict operation applied to an unfulfilled entitlement.

Independent of whether synchronisation is explicit or implicit, that synchronisation eventually terminates in one of two ways:

- ***Normal Termination*** – in the event that the reply fulfilling is received without an exceptional condition arising, the operation that triggered the synchronisation is completed and that operation returns to the caller.
- ***Abnormal Termination*** – in the event an exceptional condition is detected or received in the reply message fulfilling the entitlement, the operation that triggered the synchronisation returns abnormally, with the appropriate exception being raised.

The exception handling aspect of the entitlements mechanism is mostly straightforward, but does raise some complications, which are discussed in the following sub-sections.

7.3.2.6.1 Multiplication of Exceptions

An exception raised by a synchronous method call is raised only once, at the point at which the result of the call becomes available (which is also the point of call). The entitlements mechanism separates the point at which the result of the call becomes available from the point of call. In fact, there may be several points at which the result of a given call is made available. The exception handling mechanism described above signals exceptions at the point(s) at which the result becomes available. This raises the possibility that a given exception, corresponding to a given (failed) call, is raised more than once. This possibility is termed *multiplication of exceptions*.

Multiplication of exceptions is not the only possible semantics. The following design alternatives have been considered:

1. The exception is raised only once, in the context of the first synchronisation operation to complete on the entitlement. This has two problems:
   i. There may several copies of the entitlement, distributed across several nodes of the system. Therefore, the "first synchronisation to complete" requirement implies the establishment of a total ordering, which is certainly expensive to implement and therefore probably infeasible.
   ii. There is no obvious semantics for the second and subsequent synchronisation operations. Clearly, these cannot simply return without any indication that something is wrong. One possibility is for these operations to return a generic "result not available" exception, but this does not seem to have any advantages over multiplication of exceptions whilst having the disadvantage that information is lost.
2. The exception is raised once for each copy of the entitlement. This eliminates the requirement for a total ordering, and so is able to be implemented. However, it still suffers from the second problem noted for the previous alternative.
3. Multiplication of exceptions, as described previously, which avoids both the total ordering requirement and the need to cater for synchronisation operations in which the original exception is not raised.

The third alternative has some advantages over the other two alternatives, it has no additional disadvantages, and so is selected.

7.3.2.6.2 Sub-typing Anomaly

One result of the propagation of exceptions through entitlements is that any operation on an entitlement can raise any exception that could be raised by the remote method call associated with the entitlement. Some type systems include specifications of the exceptions that a method may raise in the rules determining validity of subtyping relationships. This interferes with the requirement that a (purely indirectly accessed) entitlement class must be a subtype of the declared return type of the method creating the entitlement. It is not at all clear that there is an elegant solution to this problem.

7.3.2.6.3 Flexibility

As described above, exceptions are handled in the context of the entitlement associated with a given request/reply pair. This corresponds to the reply-based level of exception handling context identified in the framework in Section 5.2.1.5.1. As discussed in the framework, it is desirable to provide programmers with a choice to trade-off synchronisation against depth of context.

The deepest level of context identified in the framework occurs when an exception is handled in the caller of the method that triggered that exception. The drawback of this approach is that it requires synchronisation with any outstanding calls before the caller may return. However, this depth of context is sometimes needed in order to handle a given exception effectively.

The entitlements mechanism allows exceptions relating to remote calls to be handled in the full context of the call on a call-by-call basis. In order to achieve this, the caller simply calls the *ent_wait* method for each entitlement that is to be handled in this fashion. Any exception that can arise from each such entitlement will therefore be raised before the caller returns. It would be possible to add syntactic support for this style of exception processing, but there seems little point in doing so.

7.3.3 Entitlements to Partially Directly Accessed Results

The previous section has demonstrated that entitlements for results subject purely to indirect access can be supported with minimal syntactic and semantic disruption of existing programs. In particular, existing code using remote object proxies and the results of remote method calls will continue to be syntactically correct and will, for the most part, continue to function correctly. This is achieved whilst simultaneously introducing substantial asynchrony in a controlled fashion, leading to improved performance. The primary implementation strategy for achieving this result is the use of dynamic binding of method calls with the methods of result objects that redefined to include synchronisation. This strategy is not available in the directly accessed case, and a alternative must therefore be developed.

Recall that remote objects are accessed via proxies, which are objects of classes derived from the type of the remote object. Consider an example remote object class:

```
class SERVER2 conforms REMOTE feature
  m1(p1 : INTEGER): INTEGER is …
  m2(p1 : INTEGER; p2 : CHARACTER) is …
  m3(p1 : LIST): LIST is …
  m4 : INTEGER is …
end
```

In this class, the *m1* method returns a value that is subject to direct access, an integer in this case. The *m3* method returns a purely indirectly accessed object, the *LIST* example class used previously. In a synchronous RMI system[103], the proxy class for this class will be as follows:

```
class SERVER2-PROXY conforms SERVER2 feature
  m1(p1 : INTEGER): INTEGER is …
  m2(p1 : INTEGER; p2 : CHARACTER) is …
  m3(p1 : LIST): LIST is …
  m4 : INTEGER is …
end
```

The proxy class conforms to the *SERVER2* class interface and redefines the methods of that interface to implement synchronous remote calls to methods returning non-reference values (methods returning reference values return entitlements, as discussed previously).

Objects of the proxy class given above can be accessed in the same way as local objects of the *SERVER2* class. For example

```
s2 : SERVER2 := … bind to remote class …
e : INTEGER := s2.m1(56)
s2.m2(e,'h')
s2.m2(s2.m4,'h')
```

In this example, synchronisations are required prior to the return of the calls to *m1* and *m4* since there will be no further opportunity to synchronise between the time the result is returned and the time its value is first used. In effect, the calls to *m1* and to *m4* are synchronous. This occurs because the implementation of operations on directly accessed types is fixed. However, this synchronisation is

[103] The proxy class interface for an entitlements system supporting deferred synchronisation only on indirectly accessed results will be similar, except that the return type of *m3* may be *ENT[LIST]* rather than *LIST*.

avoidable in principle, since the results produced by the calls to *m1* and to *m4* are simply used as parameters to subsequent calls to the same server, and are not used directly on the client.

Since it is not possible to remove the synchronisation of the calls to *m1* and to *m4* from the code as written above, the goal should be to remove this synchronisation with minimal modification of the code. The remainder of this section will discuss support for entitlements to directly accessed values, and how this support achieves the goal.

7.3.3.1 Implicit Conversions

To minimise the requirement for syntactic changes when entitlements for directly accessed results are used, it is desirable that values of types *ENT[X]* and *X* be interchangeable. That is, that an *ENT[X]* value can be used wherever an *X* value is expected, and vice-versa. This bidirectional equivalence requires conversions both from X to *ENT[X]* and from *ENT[X]* to *X*. In order to minimise disruption of existing code, these conversions should be implicit.

The conversion from *ENT[X]* to *X* is potentially blocking, in that the value of a directly accessed entitlement of type *ENT[X]* as a value of type *X* is not available until the reply to the call that created the entitlement has been received. The reverse conversion is non-blocking: it simply involves creating a new *ENT[X]* entitlement that is immediately fulfilled with the *X* value that is to be converted.

7.3.3.2 Proxies Revisited for Directly Accessed Results

For each remote object class with one or more method returning directly accessed values, there is an *entitlement aware proxy*, such as the following:

```
class SERVER2-ENT-PROXY feature
  m1(p1 : ENT[INTEGER]): ENT[INTEGER] is …
  m2(p1 : ENT[INTEGER]; p2 : ENT[CHARACTER]) is …
  m3(p1 : LIST): ENT[LIST] is …
  m4 : ENT[INTEGER] is …
end
```

In this proxy class, each parameter type or return type *X* that is directly accessed is replaced with the type *ENT[X]*. Parameter types of indirectly accessed values are unchanged. Observe the uniformity in the transformation of the return types, this is the reason that entitlement classes for indirectly accessed results are named in the way that they are.

This proxy can be used to achieve the desired relation of synchronisation. For example, the following will not require any reply synchronisation:

```
s2 : SERVER2-ENT-PROXY := … bind to remote class …
e : ENT[INTEGER] := s2.m1(56)
s2.m2(e,'h')                    -- The literal 'h' is implictly
s2.m2(s2.m4,'h')                -- converted to type ENT[INTEGER].
```

However, entitlements aware proxies, such as *SERVER2_ENT_PROXY* shown above, do not conform to the type of the remote object the proxy represents.

It is possible to make the entitlement aware proxy conformant by method overloading. The *SERVER2* example is as follows:

```
class SERVER2-CONFORMANT-ENT-PROXY feature
  m1(p1 : INTEGER): INTEGER is …
  m1(p1 : ENT[INTEGER]): ENT[INTEGER] is …
  m2(p1 : INTEGER; p2 : CHARACTER) is …
  m2(p1 : ENT[INTEGER]; p2 : ENT[CHARACTER]) is …
  m3(p1 : LIST): ENT[LIST] is …
  m4 : INTEGER is …
  m4 : ENT[INTEGER] is …
end
```

There are several problems with this approach:

- The versions of the proxy methods with directly accessed value types (as opposed to entitlements to such types) have synchronous semantics.
- Methods that take no (directly accessed type) parameters and return a directly accessed type value, such as m4 in the example, may not be able to be overloaded to return an entitlement for the directly accessed value type. Most language mechanisms for overloading do not allow overloaded versions of methods that differ only in return type. The reason for this is that the type of the L-value to which the return value is to be assigned is often insufficient context to perform overloading resolution to determine the method version to call.
- Consider a call to a method such as *m2* in the example, passing a mixture of directly accessed values and entitlements to such values. This call could be satisfied either by converting all actual parameters that are entitlements into directly accessed values or by converting all directly accessed value actual parameters into entitlements. In either case, the version of the method selected is that with formal parameters that match the actual parameters after the conversions have been applied. Unfortunately, the overloading resolution rules used in typical programming languages (such as C++ [ISO 1998]) will give the two possible matches equal waiting and consequently will report an ambiguity. The only way to avoid this is to define additional versions of method such as *m2*, each taking a different combination of directly accessed value parameters and entitlements. This will require 2^p versions of a method with p parameters and consequently does not scale to large numbers of parameters.

These problems render the strategy of using method overloading to maintain the conformance relationship significantly less attractive than it did at first seem, and it is therefore abandoned.

Instead, there are two proxy classes associated with each remote object class containing methods return directly accessed values. The first is the *conformant proxy*, such as the following:

```
class SERVER2-PROXY conforms SERVER2 feature
  m1(p1 : INTEGER): INTEGER is …
  m2(p1 : INTEGER; p2 : CHARACTER) is …
  m3(p1 : LIST): LIST is …
  m4 : INTEGER is …
  ent-aware-proxy : SERVER2-ENT-PROXY is …
end
```

This includes versions of all the methods of the remote object class. These behave with synchronous semantics for methods returning directly accessed value, with asynchronous semantics for methods not returning anything and with deferred synchronous semantics for methods returning values subject purely to indirect access. The conformant proxy also include a method returning a reference to the second proxy associated with the remote object class, the corresponding *entitlement aware* proxy, which is as it was originally given:

```
class SERVER2-ENT-PROXY feature
  m1(p1 : ENT[INTEGER]): ENT[INTEGER] is …
  m2(p1 : ENT[INTEGER]; p2 : ENT[CHARACTER]) is …
  m3(p1 : LIST): ENT[LIST] is …
  m4 : ENT[INTEGER] is …
end
```

All the values returned by methods of this proxy class have deferred synchronous semantics.

```
class ENT[T]
feature {NONE}
  res-buffer : T
  ctxt : PDA-ENTCTXT
feature
  ent-waiting : BOOLEAN       -- Non blocking function that returns
     is ...                   -- an indication of whether the result
                              -- has been received.
  ent-fulfilled : BOOLEAN is
     do Result := (not ent-waiting) end
  ent-wait is ...             -- Routine that blocks until the result
                              -- is received.
  conversion to T is          -- Implicit conversion to base value -
     do                       -- simply waits for the result and then
        ent-wait              -- returns it.
        Result := res-buffer
     end
  conversion from src : T is      -- Implicit conversion from base
     do                           -- value -constructs a new
        res-buffer := clone(src)  -- entitlement value,which is
        ctxt := null              -- immediately fulfilled by the
     end                          -- base value.
end
```

Figure 67 – Generic Entitlement Class for Partially Directly Access Results.

The programmer may choose between the two proxies, or may use them interchangeably. One possible approach is to modify the original code fragment as follows:

```
cs2 : SERVER2 := ... bind to remote class ...
s2 : SERVER2-ENT-PROXY := cs2.ent-aware-proxy
e : ENT[INTEGER] := s2.m1(56)
s2.m2(e,'h')
s2.m2(s2.m4,'h')
```

Apart from the extra variable declaration added at the second line, the modifications to the client code (shown underlined) are minor.

7.3.3.3 The Entitlement Generic Class

Entitlement classes for directly accessed values are instances of the parameterised class *ENT*. This is defined in Figure 67. These entitlements are represented directly by class objects, rather than by references to those class objects. In contrast to the indirectly accessed case, there is no sub-typing relationship between a directly accessed entitlement class and its base (result) type.

The *ENT* generic class defines the same explicit synchronisation operations as are defined for entitlements to indirectly accessed values. The class also defines the implicit conversion operations. These must handle a number of cases:

- An actual parameter of type *ENT*[*X*] is associated with a formal parameter of type *X*, and vice versa.
- An L-value of type *ENT*[*X*] is assigned an R-value of type *X*, and vice versa.
- Where *X* is a class type, a field or method of *X* is accessed relative to a value of type *ENT*[*X*]. This requires a conversion from *ENT*[*X*] to *X*, with the field reference or method call applied to the result of the conversion.

These may require multiple operations to be defined (conversion, assignment, field-reference) and it may not be possible to support all desirable conversions. For example, in C++, the left hand side of a member reference, *x.y* is not a candidate for implicit conversion, and it is not possible to overload the

'.' operator, so support for access to the members *X* objects relative to *ENT*[*X*] objects cannot be achieved[104].

7.3.3.4 Life Cycle

Entitlements to directly accessed values are transmitted in entitlement form only in remote calls to the responsible supplier for the entitlement. In all other remote calls, the entitlement value is converted into the corresponding base value prior to transmission, thereby potentially incurring synchronisation. This greatly simplifies the life cycle, in particular:

- There is no need to track multiple copies of a given entitlement on the primary demander or on third party nodes.
- There is no need to support remote fetching of entitlement values by third party nodes, since these node will never obtain the entitlement.

Taking advantage of relaxed synchronisation on third party nodes ad in call backs to the primary demander would require modification of the types declared in these nodes' remote object interfaces. Such major modification is almost certainly not justified. Consequently, the effective diminution of asynchrony resulting from the simplified life cycle is minimal, whilst the simplification of the system is very significant.

7.4 Assessment of Entitlements

This section assesses the Entitlement mechanism with reference to the framework developed in Chapter Five. The assessment is summarised in Table 20.

Entitlements are designed to support the construction of client/server systems and as such support the notion of a reply corresponding to a given request. Transmission of replies is an implicit result of the return of the method call associated with an Entitlement, as with RMI, there is no facility for an explicit early reply. Entitlements automate the process of reply matching and provide high level support for reply synchronisation, with the possibility of explicit reply synchronisation if it is desired. The mechanism does not restrict the kinds of values that may be sent in replies. Using Entitlements, methods with no return type are effectively one-way requests that are completely asynchronous to the caller.

The Entitlements mechanism supports coordinated termination, but does not ensure it as a consequence of the support for one-way requests described previously. The mechanism supports request forwarding automatically, although, as will be seen in the performance assessment, performance is not optimal.

Entitlements are strictly a client/server mechanism, there is no direct support for peers, and reply entailment is limited to the client/server mode. There is no support for multiple reply entailment or for reply races. Interaction structure is fixed, consisting entirely of request/reply pairs and one way requests.

The entitlements mechanism does not involve the migration of executable code between nodes. Consequently, linguistic heterogeneity is easily supported.

The user programming model of Entitlements follows that of wait-by-necessity [Caromel 1993], quite closely. Based on considerable experience with that approach, it is expected that Entitlements result in little syntactic and semantic disruption when applied to distribute existing sequential programs. Some level of semantic disruption may be inevitable in practice. In discussions with the author, Denis Caromel identified a disruption inherent in all known futures based approaches, as follows:

- There is an object *O1*, this object calls method *m2* of another object *O2*.

[104] However, C++ allows overloading of the '->' operator, so member access can be supported when *X* is a pointer to a class.

- The call to *O2.m2* immediately produces a (future) result called *f1*.
- As part of the execution of *m2*, a call to *O1.m1* is made, and *m2* can not return until the value of the result of this call is available.
- The execution of *m1* uses the value of *f1*, which is not yet available, and never becomes so, consequently, m1 never completes and the system is deadlocked.

In contrast, in a synchronous system, the execution of *m1* would use the old value of *f1*, and the interaction would complete. The situation in the system with futures can be termed the "nested future use" problem.

There is clearly a semantic disruption in this example, but it is unclear that the impact of this disruption is of much consequence in practice. Supposing that the objects in the example were monitors, the "nested monitor call" problem [Lister 1977] would occur at the point where *O2* calls *O1.m1*. With standard monitor semantics, the system would deadlock at this point. Now, it may be possible to solve the nested monitor call problem by changing the semantics of monitors in some, but such alteration will certainly add complexity and may in fact be a strategic error. Parnas [Parnas 1978] argues that instances in which the nested monitor problem occurs inevitably represent errors in the definition of interaction between system modules (monitors) and, as such, the problem is a non-problem in practice. Parnas's analysis of the nested monitor call applies equally to the nested future use problem, so if it is correct for the first instance, then nested future use is similarly a non-problem.

Entitlements are designed to operate in the context of object-oriented programming languages and to be compatible with the support for software engineering provided by those languages. The position taken in this thesis is that software engineering support in such languages is based on support for contract based programming (design by contract). Consequently, the assessment in respect of software engineering issues generally follows that for languages such as Eiffel, that directly support the contractual approach. There are two major deviations from this general policy, as follows:

- Entitlements support handling of exceptions raised in remote calls. It is important that this be done as with *at-most-once* call semantics, it is always possible, even in a correct system, that remote calls will raise exceptions.
- The issues raised by preconditions and postconditions of asynchronous remote calls are left open, since the solutions to these issues proposed for Eiffel (SCOOP) may be unsuitable for use in wide area system where minimisation of latency is a priority.

Both of these deviations are consequences of the intended use of the mechanism in wide area distributed systems, rather than general deviations from the contract based object-oriented programming model.

In the two single server interactions (*SSS* and *COSS*), Entitlements attain the optimal performance for mechanisms that do not involve code migration. This result is due to the ability to cache result value on the responsible supplier (which is the server for the interaction).

Recall from Section 5.3.1.4.3 the minimum critical path length for the *SSM* interaction is $r + r'$. This is derived by the addition to the result for a single server ($2r$) of an extra message for each straddle point between one server to the next that is not an R-group boundary, there being $r' - r$ such straddle points (and therefore $r' - r$ extra messages). These extra messages serve the following purposes:

- Propagation of result values produced on server n to server $n + 1$.
- Ensuring that the first operation executed on server $n + 1$ executes after the last operation executed on server n.

Capability	**Entitlements**
Replies	True
Early Reply	False
Reply Matching	Full support (3/3)
Reply Synchronisation	Hybrid (3/3)
Reply Kinds	All
Coordinated Termination	Supported (2/3)
Syntactic Disruption	Limited to None (2.5/3)
Semantic Disruption	Consistent with Exceptions (2/3)
One-way Requests	True
Request Forwarding	Automatic (4/4)
Peer Symmetry	False
Reply Entailment	Simple Client/Server (2/3)
Reply Races	False
Multiple Replies	False
Interaction Structure	Fixed (1/4)
Linguistic Heterogeneity	True

Capability	**Entitlements**
Type Checking Timing	Static
Type Checking Clustering	True.
System Assertion Checking	True
Assertion Coverage	Predicate (3/4)
Intrinsic Assertion Specification	True.
Inseparable Assertion Specification	Inseparable (3/4)
Declarative Assertion Specification	Propositional terms + functions (3/4)
Early Assertion Checking	Wholly dynamic (1/2)
Precondition Synchronisation	(issue left open)
Asynchronous Contract Completion	(issue left open)
Reply Specification Coverage	Simple (2/3)
Multiple Reply entailment	False.
Interaction Structure	Fixed (1/4)
Intrinsic Reply Specification	True
Inseparable Reply Specification	
Declarative Reply Specification	
Positive Reply Specification	
Reply Sending Enforcement	
Illegal Reply Handling	Prevention (4/4)
Illegal Reply Rejection Timing	Not Applicable
Extra Reply Handling	Prevention (4/4)
Extra Reply Rejection Timing	Not Applicable
Asynchronous Method Call Exception Handling	True
Exception Handling Contexts	Reply (2/4), Caller (4/4)
Exception Handling Flexibility	True

Interaction	**Entitlements**
$\mathrm{I}_{SSS}(f,r,o,n,m)$	*2r*
$\mathrm{I}_{SSM}(f,r,o,n,m,s,r',o')$	*2r′*
$\mathrm{I}_{AD}(a)$	$\begin{Bmatrix} 4 & \text{if } a=1 \\ 2a+1 & \text{otherwise} \end{Bmatrix}$
$\mathrm{I}_{COSS}(f,r,o,i,n,m)$	*2r*
$\mathrm{I}_{COSM}(f,r,o,i,n,m,s,r',o',i',ir')$	*2ir'*

Overlapped Computation	(6/7)
Request Transmission Policy	Eager (3/3)
Reply Transmission Policy	

Table 20 – Assessment Checklist– Entitlements.

The Entitlements mechanism needs to achieve the same effects, but does not do so with the minimum number of messages (one per straddle point).

Instead, the Entitlements mechanism requires two critical path messages per straddle point. The ordering constraint is achieved by an explicit wait by the client on the last operation executed on server n prior to the transmission of the first operation to server n +1. This costs one critical path message in waiting for knowledge that the last operation on server n has completed and another critical path message in transmitting the first operation to server $n + 1$. Additional messages will be required to deliver the results of other operations on server n to the client, but these will not be critical path messages as they are concurrent with the message delivering the last result. Furthermore, these message will, by definition, arrive before any operations are transmitted to server $n + 1$. Therefore, all operation requests to server $n + 1$ will contain fulfilled Entitlements and no additional messages will be required to obtain the fulfilling values of these Entitlements. Additional messages will be required to transmit the second and subsequent operations to server $n + 1$, but these will not be on the critical path, as they are concurrent with the message transmitting the first operation. Consequently, no additional critical path messages are required to propagate result values and the number of critical path messages per straddle point is two.

Given this, the total number of critical path messages is $2r + 2(r' - r)$. This simplifies to $2r'$ messages in the critical path for the *SSM* interaction using Entitlements. This is worse than the optimal result since $r' \geq r$.

Similarly, recall from Section 5.3.1.8.3 the minimum critical path length for the *COSM* interaction is found to be $r + ir'$ by similar reasoning to that used for *SSM*, and the marginal cost of inter-server result propagation and operation ordering is ir'-r. As for *SSM*, this marginal cost is doubled for Entitlements, so the overall critical path length for *COSM* is $2r + 2\ (ir'-r) = 2ir'$, also worse than optimal since $ir' \geq r$.

Analysis of the mechanism's performance for the administrator delegation interaction is complex as, like the multi-server interactions above, it involves consideration of the mechanism used to propagate results between servers. Figure 68 shows the messages sent for administrator delegation interactions with one, two and three administrators. The messages are labelled with sequence numbers starting from one, with messages that can be sent concurrently labelled with the same number. The critical path length for each interaction is the sequence number of the highest numbered message (which is always a message returning the result value to the client).

The critical path length with one administrator is four messages, with two administrators, five and with three, seven. The case with four administrators (not shown) takes the same form as that with two and three, and has a critical path length of nine messages. All cases with larger numbers of administrators also take this form. Consequently, with the exception of the single administrator special case, the critical path message size is $2a + 1$ for a administrators, only a single message better than for RPC/RMI.

Entitlements offer good support for overlapped computation on the client. Aside from actual use of a result, the only factor causing the client to block is an attempt to send an unfulfilled non-reference (partially directly accessed) Entitlement to a third party node. Request and reply transmission policy is eager by default, although this is not really part of the Entitlements mechanism and could be varied as circumstances require.

7.5 Responsibilities

Entitlements represent only the consumer side of a contract regulating the supply of a reply value. In reality, the producer of a reply is also involved in any reply contract, but with entitlements, this aspect of the contract is implicit rather than explicit.

Responsibilities [Detmold and Oudshoorn 1996a, 1996b], are a conceptual development of entitlements in which an explicit interface is available for the supplier(s) as well as for demanders.

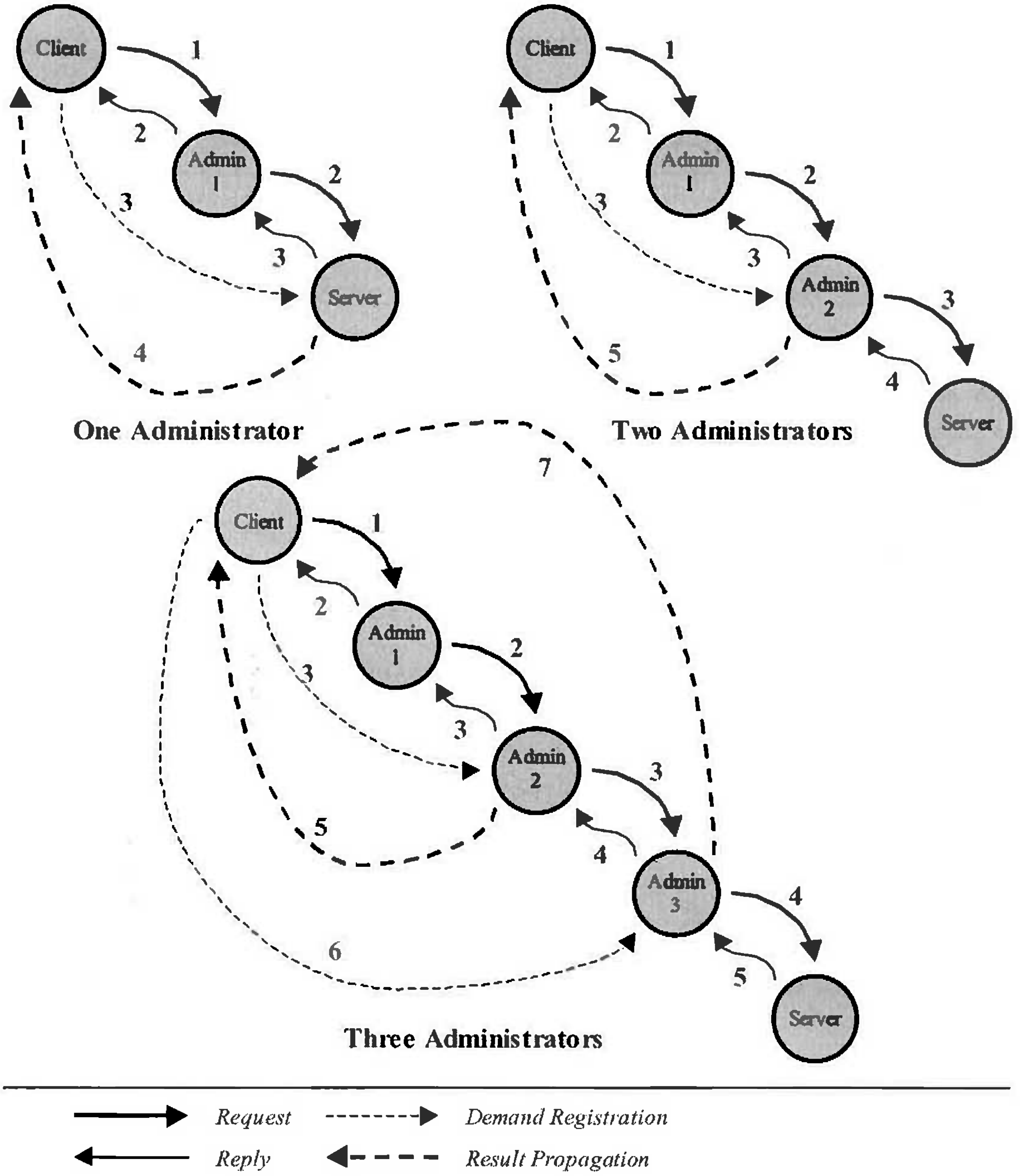

Figure 68 – Administrator Delegation with Entitlements.

The availability of this interface greatly increases the modelling power of responsibilities as a means of modelling reply contracts.

Although responsibilities can validly be regarded as augmenting the facilities provided by entitlements, it is equally valid to regard entitlements as a restricted case of responsibilities. This view, in fact, more closely models the historical development of this thesis research, in that responsibilities were actually defined prior to entitlements[105].

7.5.1 Concept and Operations

A responsibility is a type, written thus:

[105] In some previously published papers and technical reports, the term demand-only responsibilities is used instead of entitlements.

```
class RESP[T]
  inherit ENT[T]
feature
  resp-supply(fulfilling-value : T) is ...
  resp-demand : T is ...
  ent-waiting : BOOLEAN is ...
  ent-fulfilled : BOOLEAN is ...
  ent-wait is ...
  conversion to T is ...
  conversion from T is ...
end
```

Figure 69 – Operations Defined for Responsibility Types.

```
RESP[T]
```

The *RESP* type constructor produces a responsibility type from an arbitrary argument type, denoted *T* in the example.

The responsibility type can be represented by a generic *RESP*[*T*] class, the interface of this class in shown in Figure 69. Three of the operations in this interface are inherited from the corresponding *ENT*[*T*], these concern explicit synchronisation, as discussed previously. The remaining two operations support explicit supply and demand of the fulfilling value of a responsibility value. Responsibilities support both implicit demand and supply behaviour, inherited from entitlements and explicit demand and supply, via these new methods.

Figure 70 shows an example of the use of responsibilities. This example is intended purely to assist in explanation of the mechanism; in practice, one would use a mechanism such as entitlement for this example.

The example includes both client code (on the left) and server code (on the right). As with entitlements, the client interacts with the server via a proxy. The interaction between client and server proceeds in a number of steps, as follows:

1. The client creates a new responsibility, in this case a responsibility to be fulfilled with an *INTEGER* value. As with entitlements, this responsibility value is initially in the unfulfilled state.
2. The client calls a method of the server (via a proxy), passing the responsibility value as an argument to that method call.
3. The method executes on the server, accessing the responsibility via the formal parameter *res*.
4. The execution of the method calls the *resp_supply* method of the responsibility object, passing an *INTEGER* value as argument. This causes a message containing this fulfilling value to be sent back to the client (the creator of the responsibility).
5. The client executes the *resp_demand* method on the responsibility, this blocks until the fulfilling value arrives at the client, and then that fulfilling value.

This interaction between client and server is underpinned by a contract. The client should call a method with that method's pre-conditions satisfied, including passing all required actual parameters. In exchange, the server should deliver a method execution with post-conditions fulfilled, including arranging for the fulfilment of any responsibilities sent by the client.

A server might fulfil the responsibilities passed to it directly, as in Figure 70, or it might pass some or all of those responsibilities to some third party or parties. This is a sub-contractual relationship. For example, Figure 71 shows an administrator serving the client of the previous example. This administrator conforms to the interface of the server in Figure 70, and as far as the client is concerned,

```
remote-result : RESP[INTEGER]
  := new RESP[INTEGER]
remote-num : INTEGER
server : RSERVER := ...
...
server.m1(remote-result)
...
remote-num
  := remote-result.resp-demand
```

```
class RSERVER
feature
  m1(res : RESP[INTEGER]) is
     local
        num : INTEGER
     do
        ...
        res.resp-supply(num)
     end
end
```

Figure 70 – Introductory Example of the use of Responsibilities.

```
class RADMIN
  conforms RSERVER
feature
  m1(res : RESP[INTEGER]) is
     local
        server : RSERVER := pool.get-server       -- Get a server
     do                                           -- from the pool.
        server.m1(res)
     end
end
```

Figure 71 – An Administrator Server Sub-contracting Responsibilities to a Pool of Worker Servers.

is wholly responsible for fulfilling the responsibility sent. In actually, however, the administrator subcontracts the responsibility to a server selected from a pool. The selected server supplies a value for the responsibility, in exactly the same way as the server from Figure 70, sending the fulfilling value directly back to the client.

A server might also delegate a given responsibility to several other servers, creating a race between those servers as competing suppliers. In this case, each such server will independently send a supply message to the creator of the responsibility. The fulfilling value of that responsibility is the value contained in the first supply message to reach that responsibility's creator.

The implications of explicit supply in combination with *RESP* types parameterised by types containing *RESP* elements are quite profound. In particular, this combination allows extended interactions consisting of an arbitrary number of messages to be modelled using the type system. This, is a highly desirable outcome, for two reasons:

- The modelling power of programming language type systems is well developed, this development having been pursued almost since the beginning of programming.
- The use of an existing modelling domain, in this case the type system, is preferable to the introduction of an entirely new domain for modelling interactions, since the first approach minimises the increase in language complexity.

To illustrate this outcome, a detailed example using the mechanism to support a communication structure for a revision control system will is employed.

Interactions in the revision control system are initiated by, and terminate with, the client. Four messages are exchanged, as follows:

- *Check-out Request* – the client sends the server a request to check out a particular object for potential modification.
- *Check-out* – the server responds, sending a copy of the object.

```
abstract class CHECKIN-OR-ABORT
feature
  server-response : RESP[NULL] := new RESP[NULL]
end

class CHECKIN
  inherit CHECKIN-OR-ABORT
feature
  object : OBJECT;
  comment : STRING;
  constructor(ob : OBJECT, cmt : STRING) : object(ob),comment(cmt)
end

class ABORT
  inherit CHECKIN-OR-ABORT
end

class CHECKOUT
feature
  client-response : RESP[CHECKIN-OR-ABORT]
     := new RESP[CHECKIN-OR-ABORT]
  object : OBJECT;
  constructor(ob : OBJECT) : object(ob)
end
```

Figure 72 – Message Classes for Revision Control System Interactions.

- *Check-in or Abort* – the client responds, either checking-in a modified copy of the object or abort the revision. In either case, the object is unlocked and becomes available to other clients.
- *Acknowledgment* – the server responds, acknowledging the client's last message, and allowing the interaction to terminate.

Notice that these four message are conceptually part of a single interaction, but would be implemented with two request/reply pairs using conventional communication mechanisms.

Unlike conventional mechanisms, Responsibilities allow the interaction to modelled in an integrated fashion. The first step is to define a collection of types representing the continuation of the interaction past the initial message. Suitable types are shown in Figure 72, notice in particular the use of Responsibility fields to represent the continuation of the interaction at its various stages.

A server supporting the revision control interaction is shown in Figure 73. The interaction protocol is in no way dependent on this (or any other) server, another server class might implement the server side of the protocol independently of this server. The server uses an explicit type discrimination operation (the **project** statement) to deal with the two possible responses the client may send at the *check-in or abort* stage.

Finally, Figure 74 shows an example client. This creates a Responsibility, sends it to the server in a *revise* request, then operates (in some way) on the object returned via the Responsibility. Having done something with the object, the client then determines whether or not to submit a revised version, and supplies the Responsibility created by the server (in the *server-response.client-response* field) with a value appropriate to the choice made.

Another consequence of the more explicit model provided by Responsibilities (as opposed to that of Entitlements) is that it becomes possible to set up races in which two or more server compete to supply a given result. It order to set up a race, a client simply creates a Responsibility (as before) and then passes that same Responsibility to two or more remote calls. If there are two or more suppliers, then the fulfilling value of the Responsibility is the value that arrive first at the client that created the Responsibility.

```
class REVISION-CONTROL-SERVER
feature
  revise(object-id:STRING,server-response:RESP[CHECKOUT]) is
     local
        ob : OBJECT := lock-and-retreive(object-id)
        reply : CHECKOUT := new CHECKOUT(ob)
     do
        server-response.resp-supply(reply)
        project reply.client-response as CR onto
           CHECKIN :
              store-and-unlock(object-id,CR.object,CR.comment)
           ABORT :
              unlock(object-id)
        end
        reply.client-response.server-response.resp-supply(null)
     end
end
```

Figure 73 – Example Server for Revision Control System Interaction.

```
local
  server : REVISION-CONTROL-SERVER
  server-response:RESP[CHECKOUT]
  server.revise(id, server-response)
  ob : OBJECT := server-response.object
  client-response : CHECKIN-OR-ABORT := null
do
  ...
  if revised then
     client-response := new CHECKIN(ob,cmt)
     server-response.client-response.resp-supply(client-response)
  else
     client-response := new ABORT
     server-response.client-response.resp-supply(client-response)
  end
  client-response.server-response.ent-wait
```

Figure 74 – Example Client for Revision Control Interaction.

7.5.2 The Responsibilities Mechanism

The section describes the responsibilities mechanism with reference to the entitlements mechanism previously described. As would be expected from the inheritance relationship between *RESP*[*T*] and *ENT*[*T*], there is substantial commonality between the two mechanism. The major differences are as follows:

- Changes to the semantics consequent on the possibility that a Responsibility may have multiple suppliers and therefore multiple candidates for the fulfilling value.
- Additional operations supported by responsibilities.

The subsections below discuss these issues.

7.5.2.1 Consequences of Multiple Potential Suppliers

The potential for there to be multiple competing suppliers for a given responsibility has profound implications. The Entitlements mechanism ensure that there is only one supplier for a given entitlement. This is termed the *one supplier property*. An immediate consequence of the one supplier property is that it is possible to identify the (one) supplier for an Entitlement as the responsible supplier for that Entitlement. The responsible supplier for an Entitlement has authoritative knowledge of the fulfilling value for that Entitlement, since it was the responsible supplier that supplied that value. Consequently, any demands for the value can be served by the responsible supplier, and demands for the value within the responsible supplier node itself can be served without any network transactions.

Responsibilities do not have the one supplier property. There is no responsible supplier to act as the authoritative source of the fulfilling value, instead here may be several *potential suppliers* each supplying a *candidate fulfilling value*.

- Since the potential suppliers supply candidate fulfilling values asynchronously, there needs to be an arbitration mechanism that determines which of the candidate fulfilling values is the authoritative fulfilling value.
- There needs to be a mechanism to obtain the authoritative fulfilling value.

The second of these issues is dependent on the first, so it is likely that both should be addressed by a single mechanism.

In respect of the selection of the authoritative fulfilling value from the various candidate values, there are two alternatives:

- A centralised mechanism in which a single node performs the arbitration.
- A distributed mechanism, in which the authoritative value is identified by some form of consensus protocol between the potential suppliers.

The distributed alternative is inappropriate in this context, since there is no reason why each of the potential supplier should not supply a different value, and consequently there can be no guarantee of consensus being reached, even if the system is free from failures. Therefore, the centralised alternative is selected, and the authoritative fulfilling value is simply the first value to arrive at the centralised arbitrator.

There is really only one candidate for the role of centralised arbitrator: the node that created the Responsibility. As with Entitlements, this node is termed the primary demander. The identity of the primary demander is part of the state of each copy of a Responsibility, so this node is known to all other nodes that deal with the responsibility.

Having determined that the primary demander is the arbitrator of thew authoritative fulfilling value is natural that the primary demander should also server as the authoritative source of that value for nodes that demand it. In effect, the primary demander serves as the responsible supplier for the authoritative value.

Recall that in the Entitlements mechanism, there is a simple replication protocol whereby the fulfilling value of a given Entitlement is sent to third-party nodes by the responsible supplier in response to demand registration messages from third parties. Third parties cache the value they receive, second and subsequent accesses to the value at a given third party node have minimal latency. This approach is adapted for Responsibilities, as follows:

- As described above, it is the primary demander that holds the authoritative fulfilling value, and therefore is responsible for serving demand registration requests.
- All nodes other the primary demander must fetch the value of a Responsibility via the replication protocol. This includes nodes that supply a value for that responsibility, as there is no way to ensure that the locally supplied candidate value will become the authoritative value.
- With Entitlements, the substitution of the fulfilling value for a given Entitlement at the responsible supplier for that Entitlement is eager, occurs during the unmarshalling process and results in the suspension of that process until the fulfilling value is available. This is a reasonable choice for Entitlements, since it is known that the fulfilling value will be produced

locally and will be available immediately it is produced. This is not a reasonable choice for Responsibilities, as these condition do not obtain. Therefore, substitution of fulfilling value for a Responsibility on a (candidate) supplier is lazy, as it is on all other nodes.

Therefore, the mechanism is considerably less latency efficient than Entitlements in cases where there is only one supplier for a given result and the value of the result is used on that supplier node. This is an important consideration since, as has been seen, it is quite common for this situation to occur. Consequently, Responsibilities should be used only when Entitlements do not provide the necessary functionality.

7.5.2.2 Additional Operations

A Responsibility type inherits several operations from the corresponding Entitlement type and adds two new operations, *resp_supply* and *resp_demand*.

The more important of these two operations is *resp_supply*. It is this operation which supports the explicit notion of supplying a value for a result which is the major distinction between Responsibilities and Entitlements at the language level.

At the design and implementation level, the effect of an explicit supply operation on a given Responsibility is to send a supply message to the primary demander for that Responsibility. This is similar to the implicit supply operation that takes place when the method call associated with an Entitlement returns. However, in the case of Responsibilities, the supplied value is not immediately cached at the supplier, in contrast to the case with Entitlements. Instead, the supplier must obtain the fulfilling value from the primary demander. This value may be cached once obtained thus.

The *resp_demand* operation is included mainly for symmetry with the explicit supply operation. For type *RESP*[*T*] where *T* is a purely indirectly accessed type, the implicit demand and explicit synchronisation facilities provided by Entitlements are available and are usually to be preferred. For types that are partially directly accessed, the *resp_demand* operation is precisely equivalent to the implicit conversion (to T) operation defined for Entitlements to such types, and this implicit conversion operation is again usually to be preferred.

7.5.3 Advanced Example – Peer Negotiation

This section describes a detailed example of an extended interaction within a distributed system and discusses how responsibilities can be used to model and implement this interaction. The intention with this example is to demonstrate the following:

- The use of Responsibilities in combination with a programming language type system to model and implement an interaction having a complex message exchange structure.
- The use of Responsibilities to construct peer interactions, in addition to the more conventional client/server interactions described previously.

In addition, a revised version of the example is presented which eliminates any dependence of explicit type discrimination operations, replacing the use of such mechanisms with object-oriented techniques.

The application consists of an arbitrary number of nodes, interacting as peers to conduct negotiations between pairs of nodes. A node may be involved in several negotiations at once. The application is somewhat reminiscent of a Contract Net [Smith 1979].

Negotiations in the system may be initiated by any node. Each negotiation is structured into up to completion phases:

1. *Negotiation* – the peers alternate exchanging an arbitrary number of bids.
2. *Conclusion* – one of the peers either accepts or rejects the other peer's latest bid.
3. *Implementation* – in the event that the most recent bid was accepted in the previous phase, both peers implement (in some fashion) the terms agreed in that bid.

```
abstract class NEG-STEP  -- Super-class of most classes representing steps in the
   null                  -- negotiation.
end

class CURRENT-BID        -- Helper class containing information about the
feature                  -- current bid.
   constructor(b:BID) : current-bid(b)
   current-bid : BID
end

class NEG-REJECT         -- Abandon the negotiation.
inherit
   NEG-STEP
end

class NEG-CONTINUE       -- Continue the negotiation with a counter bid to which
inherit                  -- the peer should respond.
   NEG-RESP-STEP,NEG-CURRENT-BID
feature
   constructor(b:BID) : CURRENT-BID(b)
   continuation : RESP[NEG-STEP]
end

class NEG-ACCEPT         -- Accept the peer's latest bid, peer responds when
inherit                  -- implementation complete.
   NEG-STEP, CURRENT-BID
feature
   constructor(accepted:NEG-CONTINUE) : CURRENT-BID(accepted.current_bid)
   continuation : RESP[NEG-COMPLETE]
end

class NEG-COMPLETE       -- Response to acceptance of negotiation by peer, sent
   null                  -- after implementation of negotiated bid is complete.
end
```

Figure 75 – Message Classes for Negotiating Agents Interactions.

4. *Completion* – in the event that implementation was undertaken, the peer that sent the accepted bid sends a message notifying its completion of the implementation to the other peer (the peer that accepted the bid). The peer that accepted the bid acts as termination coordinator for the interaction.

The message exchange can be described by a grammar as follows:

```
continue ::= BID step
step ::= continue | accept | REJECT
accept ::= ACCEPT COMPLETE
```

The peers alternate in sending the messages represented by terminal symbols given in upper case text in this grammar.

Using Responsibilities, the structure of this negotiation can be defined by a collection of related types, as shown in Figure 75. Notice the use of responsibility types to represent the continuation of the interaction in the *NEG_CONTINUE* and *NEG_ACCEPT* classes.

Given types as defined in Figure 75, the type system enforces guarantees on message structure, as follows:

- Messages of actual type *NEG_STEP* can not be sent because this is an abstract type. Only concrete sub-types of this type can be sent.
- Replies can only be sent in response to *NEG_CONTINUE* and *NEG_ACCEPT* messages, not in response to *NEG_REJECT* or *NEG_COMPLETE* messages.
- Replies to *NEG_CONTINUE* messages may only be of the types, *NEG_REJECT* and *NEG_CONTINUE*.

```
abstract class NEGOTIATOR
feature
  implement(b:BID) is abstract
  local-response(b:BID) : NEG-STEP is abstract
  peer-continue(latest-local-step:NEG-CONTINUE) is
      do
          project latest-local-step.continuation as PS onto
              NEG-ACCEPT :
                  implement(latest-local-step.current-bid)
                  PS.continuation.resp-supply(new NEG-COMPLETE)
              NEG-REJECT :
                  null
              NEG-CONTINUE :
                  local-continue(PS)
          end
      end
  local-continue(peers-latest-step:NEG-CONTINUE) is
      local
          next-local-step : NEG-STEP := local-response(peers-latest-step.current-
bid)
      do
          peers-latest-step.continuation.resp-supply(next-local-step)
          project next-local-step as LS onto
              NEG-ACCEPT :
                  implement(peers-latest-step.current-bid)
                  LS.continuation.ent-wait
              NEG-REJECT :
                  null
              NEG-CONTINUE :
                  peer-continue(LS)
          end
      end
  start-negotiation(peers-first-step: NEG-CONTINUE) is
      do
          local-continue(peers-first-step)
      end
  send-bid(bid:BID, peer:NEGOTIATOR) is
      local
          first-local-step : NEG-CONTINUE := new NEG-CONTINUE(bid)
      do
          peer.start-negotiation(first-local-step)
          peer-continue(first-local-step)
      end
end
```

Figure 76 – Example Class of Objects using the Negotiation Protocol of Figure 75.

- Replies to *NEG_ACCEPT* messages may only be of the type *NEG_COMPLETE*

The type system cannot guarantee that extra replies are not sent, but the responsibilities mechanism will discard such extra replies, giving the programmer a static guarantee that they will not be received.

The message protocol defined by these classes is independent of the class of the objects engaged in negotiations. Indeed, the protocol might be used for negotiations between objects of unrelated types. An example class of objects using the negotiation protocol is shown in Figure 76. With this class, a negotiation is commenced by calling the *send_bid* method of one of the objects, passing an initial bid and reference to the other object. This class implementation makes use of a **project** statement to discriminate different types of *NEG_STEP* sub-types at run time. This is arguably a flaw, for reasons described previously, the use of facilities such as the **project** statement is regarded as anathema in object-oriented programming. Instead, programs should generally use polymorphic method calls.

```
abstract class NEG-STEP  -- Super-class of most classes representing steps.
feature                  -- negotiation.
   local-continue(neg:NEGOTIATOR) is abstract
   peer-continue(neg:NEGOTIATOR) is abstract
end

class CURRENT-BID         -- Helper class containing information about the
feature                   -- current bid.
   constructor(b:BID)  : current-bid(b)
   current-bid : BID
end

class NEG-REJECT          -- Abandon the negotiation.
inherit
   NEG-STEP
feature                   -- Handle NEG_REJECT steps by abandoning further action.
   local-continue(neg:NEGOTIATOR) is do null end
   peer-continue(neg:NEGOTIATOR) is do null end
end

class NEG-CONTINUE        -- Continue the negotiation with a counter bid to which
inherit                   -- the peer should respond.
   NEG-RESP-STEP,NEG-CURRENT-BID
feature                   -- Handle NEG_CONTINUE steps by call back to neg.
   local-continue(neg:NEGOTIATOR) is do neg.peer-continue(this) end
   peer-continue(neg:NEGOTIATOR) is do neg.local-continue(this) end
   constructor(b:BID)  : CURRENT-BID(b)
   continuation : RESP[NEG-STEP]
end

class NEG-ACCEPT          -- Sent to accept the peer's latest bid, peer responds
inherit                   -- when implementation complete.
   NEG-STEP,CURRENT-BID
feature                   -- Handle NEG_ACCEPT steps by implementing the bid and
                          -- then completing the interaction.
   local-continue(neg:NEGOTIATOR) is
      do
         neg.implement(current-bid)
         continutation.ent-wait
      end
   peer-continue(neg:NEGOTIATOR) is
      do
         neg.implement(current-bid)
         continuation.resp-supply(new NEG-COMPLETE)
      end
   constructor(accepted:NEG-CONTINUE)  : CURRENT-BID(accepted.current_bid)
   continuation : RESP[NEG-COMPLETE]
end

class NEG-COMPLETE        -- Response to acceptance of negotiation by peer, sent
   null                   -- after implementation of negotiated bid is complete.
end
```

Figure 77 – Negotiation Message Classes Incorporating Behaviour.

It may be possible to adopt the polymorphic method call approach by embedding behaviour in the message classes, as shown in Figure 77. The constraint on this approach is that it must be possible to transport this behaviour between nodes on the system. This may be a problem if nodes are implemented in different programming languages.

The major consequence of embedding behaviour in the message classes is a significant reduction in the complexity of classes of objects using the message protocol. The abstract class of such objects is shown in Figure 78. This class is functionally equivalent to the class given in Figure 76. Note however, that in contrast to the previous case, all objects using the protocol must be sub-types of the type of this class, because there are dependencies from the message classes on the *NEGOTIATOR* type.

7.6 Assessment of Responsibilities

This section assesses the Responsibilities mechanism with reference to the framework developed in Chapter Five. The Responsibilities mechanisms is best assessed in comparison to the entitlements mechanism. Table 21 on page 236 shows the assessment checklist results for Responsibilities and also includes the assessment checklist results for Entitlements to assist comparison.

The major advantages that the Responsibilities mechanism has in comparison to the Entitlements mechanism are as follows:

- Support for communication between peers as well as for client/server systems.
- Support for multiple replies in response to a given request, and for races between suppliers of a given reply to a given request.
- Support for interaction structures that extend past a single request/reply pair, in both client/server and peer contexts. These interaction structures are modelled in the host language's type system, and hence can be recursive in nature (given that the host's type system is so).

The disadvantages are as follows:

- Syntactic disruption of the server (which must execute an explicit supply operation).
- Filtering of extra replies, as opposed to rejection at the source.
- Diminished performance in many cases, as described further below.

Importantly, the two mechanisms are integrated, via a subtype relationship between corresponding Responsibility and Entitlement types. Consequently, it is possible to pick the better mechanism for a given task on a case by case basis.

The diminished performance of Responsibilities is an inevitable consequence of the loss of the one supplier property that was present with Entitlements. This, in turn, is a consequence of the increased flexibility afforded by Responsibilities.

The loss of the one supplier property means that the fulfilling value of a given Responsibility can only be obtained from the node that created that Responsibility. Consider the case of a server executing a group of result interdependent operations (an R-group) responsible for supplying values for a collection of responsibilities created by a given client. The requirement to obtain fulfilling values from the client means that there must be a two message round trip back to the client between every result independent sub-group (O-group) within the R-group, in the critical path of the interaction. This means that the number of critical path messages for the R-group is $2 + 2x = 2\ (x + 1)$, where x is one less than the number of O-groups in the R-group. Simplified, this is twice the number of O-groups in the R-group, so the critical path message length for an entire sequential interaction is simply twice the number of O-groups in the interaction , that is $2o$ for the *SSS* interaction. The critical path is also $2o$ for the single server concurrent interaction (*COSS*), since any concurrency is contained within O-groups.

The *SSM* interaction requires $2o$ messages as for *SSS*, plus extra messages to provide ordering and result propagation across straddle points between servers that are not O-group boundaries (which have already been counted in the $2o$ term). As was the case with the analysis of *SSM* for Entitlements, each such straddle point requires 2 critical path messages. However, for Responsibilities, there are $o' - o$ straddle points, so the total number of critical path messages is $2o + 2(o' - o) = 2o'$ messages.

Analysis of the *COSM* case for Responsibilities requires the definition of a new kind of operation group, the IO′-group, analogous to the IR′-group defined in Section 5.3.1.8. The IO′-group is derived from the combination of O'-groups (O-groups wholly within one server) and I-groups by the following constraints:

IO′-1. A contiguous group of one or more I-groups wholly contained in an O′-group is an IO′-group.

```
abstract class NEGOTIATOR
feature
  implement(b:BID) is abstract

  local-response(b:BID) : NEG-STEP is abstract

  peer-continue(latest-local-step:NEG-CONTINUE) is
     do
        latest-local-step.continuation.peer-continue(this)
     end

  local-continue(peers-latest-step:NEG-CONTINUE) is
     local
        next-local-step : NEG-STEP := local-response(peers-latest-step.current-
bid)
     do
        peers-latest-step.continuation.resp-supply(next-local-step)
        next-local-step.local-continue(this)
     end

  start-negotiation(peers-first-step: NEG-CONTINUE) is
     do
        local-continue(peers-first-step)
     end

  send-bid(bid:BID, peer:NEGOTIATOR) is
     local
        first-local-step : NEG-CONTINUE := new NEG-CONTINUE(bid)
     do
        peer.start-negotiation(first-local-step)
        peer-continue(first-local-step)
     end
end
```

Figure 78 – NEGOTIATOR Abstract Class for use with Behavioural Message Classes from Figure 77.

IO′-2. Each I-group wholly or partially containing more than one O′-group is an IO′-group.

IO′-3. The number of IO′-groups within an O-group is minimal, subject to the previous constraints.

The number of IO′-groups within an interaction is given by the parameter io'. The number of messages in the critical path for a *COSM* interaction is found as follows:

- Assuming all operations were executed by the same server, the number of messages required is $2o$ (as given previously).
- However, the interaction spans multiple servers. Two additional messages for each straddle point between each adjacent pair of IO′-groups within an O-group to enforce the ordering of that pair.
- The number of these pairs in the interaction is the difference between the number of IO′-groups and the number of O-groups, that is $io'-o$.
- Therefore, the number of messages in the critical path is $2o + 2(io'-o) = 2io'$.

A similar approach to that in Section 5.3.1.8.2 demonstrates that $io' \leq o'$ and the critical path length for *COSM* is therefore a slight improvement on the *SSM* case.

In contrast to the other interactions, the performance for administrator delegations is actually better with Responsibilities than it is with Entitlements. In fact, Responsibilities deliver the optimal performance for this interaction.

Finally, in respect of the other performance factors, Responsibilities also attain the highest level results in the checklist. The client is only blocked when attempting actual (strict) use of an unfulfilled Responsibility, and both request and reply transmission policies are eager.

Capability	**Entitlements**	**Responsibilities**
Replies	True	
Early Reply	False	
Reply Matching	Full support (3/3)	
Reply Synchronisation	Hybrid (3/3)	
Reply Kinds	All	
Coordinated Termination	Supported (2/3)	
Syntactic Disruption	Limited to None (2.5/3)	Limited (2/3)
Semantic Disruption	Consistent with Exceptions (2/3)	
One-way Requests	True	
Request Forwarding	Automatic (4/4)	
Peer Symmetry	False	True
Reply Entailment	Simple Client/Server (2/3)	Extended Peer (3/3)
Reply Races	False	True
Multiple Replies	False	True
Interaction Structure	Fixed (1/4)	Recursive (3/4)
Linguistic Heterogeneity	True	

Capability	**Entitlements**	**Responsibilities**
Type Checking Timing	Static	
Type Checking Clustering	True.	
System Assertion Checking	True	
Assertion Coverage	Predicate (3/4)	
Intrinsic Assertion Specification	True.	
Inseparable Assertion Specification	Inseparable (3/4)	
Declarative Assertion Specification	Propositional terms + functions (3/4)	
Early Assertion Checking	Wholly dynamic (1/2)	
Precondition Synchronisation	(issue left open)	
Asynchronous Contract Completion	(issue left open)	
Reply Specification Coverage	Simple (2/3)	
Multiple Reply Entailment	False	True
Interaction Structure	Fixed (1/4)	Recursive (3/4)
Intrinsic Reply Specification	True	
Inseparable Reply Specification		
Declarative Reply Specification		
Positive Reply Specification		
Reply Sending Enforcement		
Illegal Reply Handling	Prevention (4/4)	
Illegal Reply Rejection Timing	Not Applicable	
Extra Reply Handling	Prevention (4/4)	Filtering (2/4)
Extra Reply Rejection Timing	Not Applicable	
Asynchronous Method Call Exception Handling	True	
Exception Handling Contexts	Reply (2/4), Caller (4/4)	
Exception Handling Flexibility	True	

Interaction	**Entitlements**	**Responsibilities**
$\mathrm{I}_{SSS}(f,r,o,n,m)$	$2r$	$2o$
$\mathrm{I}_{SSM}(f,r,o,n,m,s,r',o')$	$2r'$	$2o'$
$\mathrm{I}_{AD}(a)$	$\begin{Bmatrix} 4 & if\ a=1 \\ 2a+1 & otherwise \end{Bmatrix}$	$a+2$
$\mathrm{I}_{COSS}(f,r,o,i,n,m)$	$2r$	$2o$
$\mathrm{I}_{COSM}(f,r,o,i,n,m,s,r',o',i',ir')$	$2ir'$	$2io'$

Overlapped Computation	(6/7)	(7/7)
Request Transmission Policy	Eager (3/3)	
Reply Transmission Policy		

Table 21 – Assessment Checklist– Responsibilities.

Chapter Eight

Ambassadors

The framework developed in Chapter Five is rather strict in assessing middleware support for linguistic heterogeneity. Namely, the middleware is linguistically heterogeneous if the multitude of different languages providing the computational elements of the system have equal, or approximately equal, access to the middleware facilities. Furthermore, this thesis is uncompromising in requiring linguistic heterogeneity: mechanisms requiring linguistic homogeneity are not acceptable in large scale systems, which typically are built by composition of existing smaller systems and consequently in which linguistic heterogeneity is the norm.

This insistence on linguistic heterogeneity apparently excludes the use of communication techniques involving code migration. These techniques offer the possibility of attaining the ideal critical path message count for many classes of interactions, as has been demonstrated in Chapter Six. Consequently, the insistence on linguistic heterogeneity appears to have a considerable cost in performance terms.

In actuality, however, the inability to use communication techniques involving code migration comes from a restrictive definition of middleware. As traditionally defined, middleware is fundamentally passive. The aim of traditional middleware is to be semantically neutral. Given a call submitted by a client the middleware should perform an invocation on the server targeted by that operation. This call, made by the middleware on the client's behalf, should be semantically equivalent to the call the client submitted.

In contrast to passive middleware, *active middleware* is not semantically neutral. Rather than submitting individual remote operations to the middleware, clients in an active middleware system submit descriptions of complete interactions, typically including several interrelated operations. The middleware is then responsible for effecting the desired communication.

The addition of an active component to the middleware dramatically alters the situation *vis-à-vis* linguistic heterogeneity and code migration. Migration of the computational elements of the system is prevented by the requirement for linguistic heterogeneity, but nothing stops the middleware from using code migration internally to effect communication. The only requirement is that the various languages in the system continue to have approximately equal access to the middleware facilities, including any facilities related to code migration.

This chapter presents an approach to active middleware. The essential idea is the use of mobile *Ambassador* objects as agents supporting complex interactions between the computational elements of the system [Detmold, Hollfelder and Oudshoorn 1999, Hollfelder, Detmold and Oudshoorn 1999].

8.1 Activating Middleware

In order for middleware to be made active, it is necessary to add a mechanism to allow user programs (typically clients) to submit descriptions of complex interactions, which shall be termed *scripts*. The middleware system is responsible for executing scripts. To realise performance gains, the middleware may need to execute various different parts of the script on arbitrary nodes involved in a given interaction. Consequently, the middleware needs to support an execution environment for scripts on all nodes in the system. Furthermore, scripts may be complex: to achieve optimal critical path

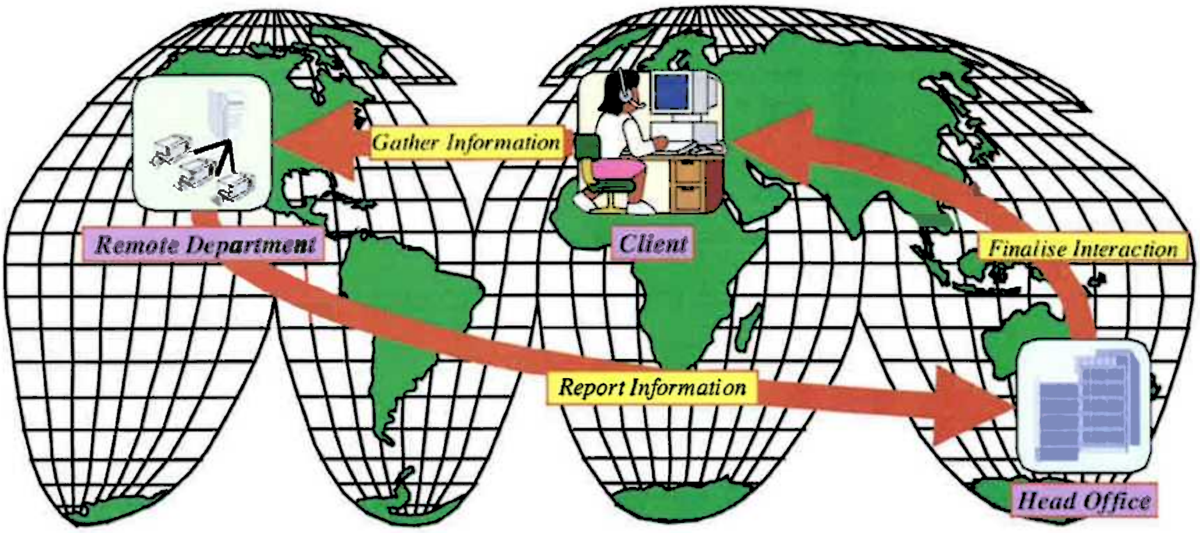

Figure 79 – An Example Worldwide Interaction.

message counts, a script may need to produce a remote operation input parameter from an arbitrary function of previously calculated remote operation results.

While it would be possible to devise a new language for scripting communication, the use of an established language such as Java [Arnold and Gosling 1997] in this role is a far better approach. Consider:

- Java runs on most common computer systems. Consequently, the middleware can simply make use of existing Java execution facilities to execute scripts, rather than requiring the development of a new execution environment.
- Java executable code can be migrated between systems of different kinds whilst maintaining the same execution semantics.
- Java is a Turing-complete programming language. As such, it is sufficient to express scripts of arbitrary complexity.

There is nothing fundamental in the nature of Java that makes it particularly well suited for this task. Instead, the motivation to coopt Java in this way stems solely from the language's ubiquity.

8.2 An Example

An example world-wide interaction is shown in Figure 79, this example operates in the context of three LANs, a client LAN, a remote department LAN and a head-office LAN. These LANs are distributed worldwide, and messages between processes on different LANs are assumed to take 200 milliseconds to propagate.

The remote department LAN contains a resource directory object and several objects representing printers currently allocated to that department. The head office LAN contains an object maintaining a record of printer usage for various departments. These records are used to assess compliance with an enterprise-wide initiative for paper-free business, rather than for accounting. Consequently, they are gathered by an *ad hoc* process initiated in a third LAN (the client LAN) rather than from the head office.

An outline of the algorithm used by the client to perform the interaction is shown in Figure 80. This figure is annotated with dependencies between remote operations. Only result and functional dependencies are shown, as there are no pure order dependencies in this interaction. Furthermore, only direct dependencies are shown: dependencies are transitive, so, for example, the call to *submitPUReport* is functionally dependent on the call to *getPrinterCount* via two intermediate functional dependencies and one intermediate result dependency.

```
numPrinters := resDir.getPrinterCount()
orgUnit := resDir.getOrgUnitName()
for printer in 1..numPrinters loop
   printers[printer] := resDir.getPrinter(printer)
   usage[printer] :=
      resDir.getUsage(printers[printer])
end
total := 0
for printer in 1..numPrinters loop
   total := total + usage[printer]
end
headOffice.submitPUReport(orgUnit,numPrinters,total)
```

Result Dependency *Functional Dependency*

Figure 80 – Algorithm for Example Interaction.

The figure identifies four dependencies. The call(s) to *getPrinter* are functionally dependent on the call to *getPrinterCount* because the execution of a given call to *getPrinter* is conditional on the result of the call to *getPrinterCount.* The call to *getUsage* in each iteration of the first loop is result dependent on the call to *getPrinter* in the corresponding iteration of that loop. Finally, the call to *submitPUReport* is functionally dependent (via the second loop) on the results of all the calls to *getUsage* and, in addition, is result dependent on the call to *getOrgUnitName.*

Now, supposing that the remote departmental LAN currently has twenty printers, one can model the performance of various communication mechanisms in terms of the number of worldwide message transmission required.

8.2.1 Analysis According to the Framework

Using the technique developed in Section 5.3.1, the group structure of the example interaction for p printers is shown in Table 22. From the diagram, the values of the parameters defined in the performance model are as follows:

$f = 1$, $r = 3$, $o = 4$, $i = 4$, $n = 2p + 3$, $m = 0$, $s = 2$, $r' = 3$, $o' = 4$, $i' = 4$ and $ir' = 3$.

Using the formulae for critical path message count for *COSM* interactions from the survey in Chapter Six, the message counts predicted by the framework for a variety of communication mechanisms are shown in Table 23.

8.2.2 Ad hoc Analysis

This section presents an ad hoc analysis of the performance of the example as a verification of the approach from the framework.

8.2.2.1 RPC/RMI

For RPC/RMI [Birell and Nelson 1984, Sun 1997a] with a single threaded client, each remote operation requires two worldwide message transmissions, and all messages must occur serially. There is one call to each of *getPrinterCount*, *getOrgUnitName* and *submitPUReport*, and (with p printers), p calls to each of *getPrinter* and *getUsage*, giving $2p + 3$ remote calls and $4p + 6$ serial worldwide message transmissions.

<table>
<tr><th>Operations</th><th>F-groups</th><th>R-groups</th><th>O-groups</th><th>I-groups</th><th>IR′-groups</th><th>I′-groups</th><th>O′-groups</th><th>R′-groups</th><th>S-groups</th></tr>
<tr><td>resDir.getPrinterCount
resDir.getOrgUnitName</td><td rowspan="4">1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td rowspan="3">1</td></tr>
<tr><td>resDir.getPrinter(1)
...
resDir.getPrinter(p)</td><td rowspan="2">2</td><td>2</td><td>2</td><td rowspan="2">2</td><td>2</td><td>2</td><td rowspan="2">2</td></tr>
<tr><td>Printers[1].getUsage
...
Printers[p].getUsage</td><td>3</td><td>3</td><td>3</td><td>3</td></tr>
<tr><td>HeadOffice.submitPUReport(...)</td><td>3</td><td>4</td><td>4</td><td>3</td><td>4</td><td>4</td><td>3</td><td>2</td></tr>
</table>

Table 22 – Group Structure of Example Interaction.

Mechanism	**Formula**	**Message Count**
RPC/RMI	$2(n + m)$	$4p + 6$
Futures	$\mathrm{Min}(2i, 2o')$	8
Multi-threaded RPC	$2i$	8
Batched Futures, Entitlements and Responsibilities	$2r'$	6
Mentat	$r + ir'$	6
Ideal	$f + s$	3

Table 23 – Critical Path Message Counts for the Example Interaction.

8.2.2.2 Multi-Threaded RPC/RMI and Futures

If the client is multi-threaded (and also if it uses a futures or promises mechanism) then the calls to *getPrinterCount* and *getOrgUnitName* can proceed concurrently. More importantly, all of the calls to *getPrinter* can proceed concurrently, as can all the calls to *getUsage*. This can be achieved by using a thread for each iteration of the first loop.

Each group of concurrent calls contributes only two worldwide transmissions to the overall latency. There are four such collections of calls (the call to *submitPUReport* is in a group by itself). Therefore, the aggregate contribution to the critical path message count is eight messages, a speedup of ten times for the case of twenty printers.

8.2.2.3 Batched Futures and Responsibilities

Batched futures [Bogle and Liskov 1994] and responsibilities [Detmold and Oudshoorn 1996a, 1996b] give further improvement, in that the calls to *getUsage* may be in progress concurrently with the calls to *getPrinter*. This reduces the number of collections of concurrent calls to three and consequently the critical path message count to six messages.

8.2.2.4 The Ideal Latency

To achieve the ideal aggregate latency, the interaction should be structured as follows:

1. Migrate code for the interaction from the client LAN to the departmental LAN. This costs one worldwide message transmission.
2. Execute the loops and all the calls, except for the final call to *submitPUReport*. This does not require any worldwide message transmissions.

3. Migrate the continuation of the interaction to the head office LAN. This costs one worldwide message transmission.
4. Execute the call to *submitPUReport*. This does not require any worldwide message transmissions.
5. Return the continuation to the client. This requires one worldwide message transmission. This step is necessary in order that the client is able to determine that the interaction has completed successfully.

That is, only three worldwide message transmissions are required in the ideal critical path. With twenty printers, this gives a speedup of nearly thirty times over the single-threaded RPC/RMI case and nearly three times over the multi-threaded case. The critical difference with this approach is that it involves the migration of code.

8.3 Ambassadors

The Ambassadors system [Detmold, Hollfelder and Oudshoorn 1999, Hollfelder, Detmold and Oudshoorn 1999] is a mechanism for structured communication in worldwide systems based on the migration of code attached to mobile objects.

Human ambassadors served an essential function in international diplomacy in the age prior to electronic communication. These ambassadors have the following attributes:

- The ambassador is located within the foreign state, communication between that foreign state and the ambassador is rapid. Consequently, issues that arise between the ambassador's home state and the foreign state are able to be resolved expeditiously. This stands in contrast to the situation if the two states needed to communicate directly via pre-electronic communication with a latency of several weeks or even months.
- A well-chosen ambassador is a local expert, with specific knowledge of the foreign state. This specialist knowledge is employed to represent the home state to the foreign state more effectively than could someone located in the home state.
- An ambassador is a plenipotentiary, acting for the home state with the full powers of that state.
- As a plenipotentiary, the ambassadors can form agreements and contracts on behalf of that state.
- An ambassador may be called on to continue representing the home state even when the lines of communication back to the home state that do exist are disrupted.
- However, whilst invested with considerable powers by the home state, the ambassador is not a free agent. It would hardly be acceptable for ambassador engaged to represent the interests of some state to spend significant portions of time pursuing private business. Instead, the ambassador is charged to pursue the business of the home state.
- One aspect in which an ambassador is directed by the home state is in the structuring of the ambassadors itinerary. Typically, an ambassador has a "tour-of-duty" of finite duration, followed by a return to the home state to report on activities undertaken.
- The world being what it is, tragedy sometimes strikes and ambassadors are killed or lost "in the line of duty", and fail to return from their tours. In such cases, the home state has little alternative but to engage another ambassador.

The Ambassadors communication mechanism is closely analogous to human ambassadors with attributes as described above.

In order to execute a batch of inter-dependent operations against a remote server, a client packages those operations into an Ambassador, which is then dispatched to the location of the server. Upon arrival at the server, the Ambassador executes the remote operations and then returns to the client, delivering the results. An Ambassador in the communication mechanism has attributes corresponding to those of the human ambassador:

- Calls from the Ambassador to the server(s) execute with low latency. This acts to expedite the processing of groups of functionally interdependent calls to that server. This is analogous to the way the human ambassador's presence in the foreign state expedites diplomatic relations.
- The interaction with the server(s) might involve detail that is irrelevant to the results desired by the client. This detail is hidden within the Ambassador and is abstracted from the client's point of view. This is analogous to the benefits deriving from local expertise of human ambassadors.
- An Ambassador represents the client in the location of the server. As such, the Ambassador operates within the security environment applying to the client, and has the client privileges or some significant subset thereof. This is analogous to a human ambassador as a plenipotentiary.
- As we have seen previously in this thesis, calls to methods of objects are governed by contracts, with the (programmer of the) caller forming a contract with the (programmer of the) called object. Ambassadors therefore form contracts with the server(s) with which they interact, on behalf of the client. This is analogous to the ability for human ambassadors to form contracts on the behalf of the home state.
- An Ambassador may reach the server(s) with which it is to interact and discover that the network has partitioned and that it is temporarily cut off from the client. It is still able to pursue its tasks, possibly adopting a strategy to counter act the partitioning, possibly just completing its tasks and then waiting to return to the client. The analogy here is the ability for human ambassadors to continue to operate when communications with the home state are disrupted.
- The ambassador exists to effect communication with the server(s), as instructed by the client. It is not a vehicle for arbitrary computation. The analogous case with human ambassadors is that they do not pursue their own agenda, but rather that of their home states.
- At the completion of interaction with the server(s), the Ambassador returns to the client. Hence, the Ambassador's tour is a closed loop, in much the same way as a human ambassador's tour of duty.
- Failure is an inescapable characteristic of most distributed systems. This can result in an Ambassadors being lost whilst on tour and consequently never returning to the client. The analogy to the case with human ambassadors here is obvious.

Recognition of these characteristics informs the semantics of the Ambassadors mechanism, which is now discussed.

8.4 Ambassadors Semantics

Ambassadors provide a means whereby a method is invoked against a local (Ambassador) object, but that object is migrated so that the invocation actually takes place on some remote node (in this case nodes are Java virtual machines). Typically, the remote node would be close to the node containing server objects with which the method called against the Ambassador was to interact.

The operation causing a method invocation to occur against a given object, but on a remote node is called a *migrate-and-invoke* operation. It entails sending a message to the node where the invocation of the object is to take place. This message contains (the transitive closure of) the state of the object, an indication of the method to be invoked and the parameters to pass to that invocation. The node receiving the message executes the method as appropriate. This execution can of course contain further migrate-and-invoke operations.

Two kinds of migrate-and-invoke operations are provided in the current Ambassador system. These are:

```
visit(REMOTE-OBJECT, AMBASSADOR, METHOD-ID, ARRAY[OBJECT])
```

```
migrate(REMOTE-OBJECT, AMBASSADOR, METHOD-ID, ARRAY[OBJECT])
```

In each case, the first parameter is a reference to a proxy for an object on a remote virtual machine where the invocation is to take place. The second parameter is the object to migrate (which must be of a class inheriting from the *Ambassador* class). The third parameter is the method to invoke against this object when it reaches the remote node. Finally, the fourth parameter provides the parameters to pass to the remote invocation.

The *visit* operation passes a copy of the object and parameters (including the respective closures) to the remote virtual machine, validates that the invocation is type-safe and, all being well, starts the invocation executing. This invocation then has low-latency access to objects resident in the remote virtual machine, via standard (local) Java method invocations. The thread that initiates the *visit* operation waits for an acknowledgment indicating either that the invocation was started successfully or that it could not be started due to a type error. Importantly, this acknowledgment is sent by the remote node concurrently with the execution of the remote invocation. If the invocation starts executing successfully then the initiating thread is suspended, otherwise the *Ambassador_Invocation_Error* exception is raised in the caller of the *visit* operation.

Initiating a *visit* operation entails an eventual reply. This reply contains an updated copy of the object that was migrated. When the reply is received, the thread that issued the *visit* operation is resumed. This thread then copies the state of the updated copy over the state of the local copy of the Ambassador object. The thread then returns normally from the call to *visit*. Ambassador objects are implicitly monitors, hence the copy of an Ambassador object left behind during a visit is inaccessible until the visit returns.

In the normal course of events, the acknowledgment that the invocation associated with a visit has started successfully would arrive at the visit's origin node before the reply from that visit. If the reply arrives first, processing is delayed until the acknowledgment has also arrived. It would be possible for an optimised implementation to treat such a reply as an implicit acknowledgment in this case and to discard the explicit acknowledgment when it arrived.

The *migrate* operation differs from *visit* in that the thread initiating a migrate operation terminates (after establishing that the remote invocation has been started successfully) instead of being suspended. Consequently, no reply is expected for a *migrate* operation and no copy back of the object occurs.

Reply messages are sent when remote invocations return. The reply message is sent to the point of the most recent *visit* operation. If the execution of a remote invocation initiated by a *visit* performs a *migrate* operation, then that invocation does not return (and does not send a reply). Instead, the invocation resulting from the *migrate* operation is responsible for the reply: either it is sent on return of the method called in that invocation or that invocation performs a further *migrate* operation.

The first *migrate-and-invoke* operation on an Ambassador is required to be a *visit* operation. This ensures that the client (which initiates the *visit*) is informed when the interaction implemented by the "tour" undertaken by the Ambassador is complete. The tour of an Ambassador is therefore guaranteed to be a cycle, completed by a reply message, potentially with sub-cycles hanging off the main cycle; and recursively so.

Figure 81 shows a simple tour in which the invocation effected by the initial *visit* (1) to a remote server executes a *migrate* operation (2), migrating to a second remote server. The invocation effected by this *migrate* operation performs a *visit* (3) to a third remote server, which returns, sending a reply

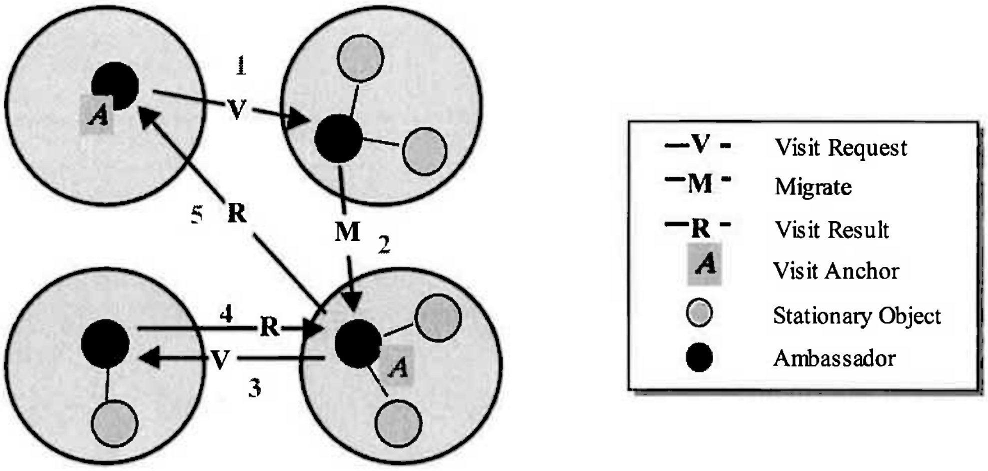

Figure 81 – A Simple Ambassador Tour.

to the second server (4). The invocation resulting from the *migrate* operation then returns, completing the tour with a reply (5) to the initial *visit.*

The copies of the object which are left behind whilst a *visit* is in progress are called *anchors*. These anchors will eventually be updated when the reply to the *visit* is received. Attempts to perform migrate or visit operations on anchors are not permitted and raise an exception. These semantics ensure there is only one "live" copy of an Ambassador at any given time. Hence, the flow-of-control of an Ambassador tour is sequential and consequently is easily understood.

The result(s) of a *visit* can be obtained by the execution of local methods defined by the Ambassador subclass. Since, as described previously, Ambassadors behave as monitors, and the monitor lock is held by a blocked thread for the duration of a visit, these accesses are blocked until the visit has returned.

8.4.1 Structure

In a distributed system there is a need for the client to obtain information about the success or failure of each Ambassador it dispatches. This implies that there must be communication back to the client from the point of occurrence of the causally last action taken by the Ambassador on its tour. The Ambassadors mechanism therefore imposes a structural constraint requiring Ambassador tours to be cycles, with Ambassadors ending up finally at the node from which they were dispatched. The structural constraint in recursive so that there may be sub-cycles attached to the main cycle.

This leads to a communication structure that is the same as that of RPC/RMI. For RPC/RMI without request forwarding, the communication structure is a main cycle (the top level call) of length two. There may be one or more sub-cycles (nested calls from the top-level call), these will also be of length two. RPC/RMI with request forwarding the has exactly the same structure as Ambassadors: a main cycle with possible sub-cycles, with all cycles being of length greater than or equal to two. The structural constraint is enforced by a requirement that the first operation in an Ambassador's tour must be a visit. Consequently, a migrate operation can only occur within the context of a visit operation, in much the same way as request forwarding can only occur within the context of a request.

8.4.2 Semantics in the Presence of Failures

Ambassador tours have *at-most-once* semantics [Birell and Nelson 1984]. The first migrate-and-invoke operation performed on an Ambassador is required to be a *visit* operation. If the Ambassador returns from this visit successfully, then the tour executes *exactly-once*, otherwise the outcome is indeterminate. Conceptually, an Ambassador tour can be viewed as a remote procedure call with at-most-once semantics. Possible failure of a remote procedure call is detected when a time-out expires before the reply from the remote call is received at the client.

The possible failure of a visit can be detected in exactly the same way as described above for a remote procedure call. In addition to this time-out-based detection of the failure of the entire tour, it is also possible to use extra time-outs to detect the failure of visits within the tour. This has the advantage of enabling more localised detection of failures and consequent early recovery. As described previously, the first migrate-and-invoke operation on any Ambassador is required to be *visit*. Consequently, any *migrate* operation is contained within a *visit*, and a failure occurring in that *migrate* operation is detected as failure of the containing *visit*.

As with at-most-once RPC [Birell and Nelson 1984], it is possible failure that is detected, not certain failure. It is possible for a visit to succeed but to have previously returned an indication of possible failure (a false negative). It is however, guaranteed that a visit will not falsely indicate a successful outcome. A principle aim of an implementation is to reduce the probability of occurrence of such false negatives. However, because false negatives can occur, it is not possible to issue retries of possibly failed visits automatically. Automatic retry would give Ambassadors at-least-once semantics, which is likely to be undesirable in the majority of cases, although the system might be extended to give programmers a choice.

8.4.3 Placement

The key idea underpinning Ambassadors is to avoid repeated high latency remote method calls by migrating the source of those method calls (an Ambassador) close to the destination of the calls. There are two choices for the placement of an Ambassador on a stage of its tour:

- The Ambassador may be *entrenched* in the node (Java virtual machine) containing the server(s) with which it is interacting and interacts with those servers by local method calls. In this mode, calls to these servers execute with optimal performance and have exactly-once semantics (from the point of view of the Ambassador).
- The Ambassador may be *hosted* in a node that is near to the node(s) containing the server(s) with which it is interacting. In this context, near means that the Ambassador is located in a node on the same computer, or in the same local area network as the server(s). In fact, the hosting node might be one of the nodes containing the server objects, the critical distinction being that calls are executed via RMI, even to a server in the same node. In this mode, the calls made by the Ambassador have the same semantics as calls made by the client via RMI, the only difference is the reduced latency.

It is not clear which of the two alternatives is to be preferred overall, so the mechanism allows both. The choice between the two alternatives is made at the server object level: when a server object is exported, it can be designated as entrenchable, in which case calls from Ambassadors running locally are local method calls. An entrenchable object continues to be accessible via RMI to Ambassadors running on different nodes and to ordinary RMI clients.

8.4.4 Security

An Ambassador acts as the representative of the client that dispatched it. It should act as intended by that client and with the same privileges as that client. The semantics of the security environment in which Ambassadors operate are as follows:

- The system must verify that the code that an Ambassador executes is as intended by the client that sent out the Ambassador.
- The system must ensure that an Ambassador acts with (at most) the same privileges as the client that sent it out.

This semantics level view of the security environment is further elaborated in the discussion of design issues in Section 8.5.

8.5 Design

The Ambassadors system makes use of the Java RMI API [Sun 1997a], for locating remote virtual machines and for transport of mobile objects. It uses the Java Core Reflection API [Sun 1997b], to effect type-safe method invocations on remote objects once they arrive at a destination in their tour. The implementation requires no modification of the Java run-time system or compiler.

8.5.1 Architecture

Users define Ambassadors as classes extending the *Ambassador* class. This class also acts as the interface through which the system manipulates Ambassadors. *Ambassador* is a class rather than an interface mainly for convenience, it would be possible to allow users the alternative of defining Ambassadors by implementing a specific interface rather than extending to *Ambassador* class. These two alternatives would then be similar to the two alternatives provided for creating remote object with RMI.

The Ambassadors system has two servers running at every node, these are responsible for transporting Ambassadors around the system and initiating invocation of methods requested on Ambassador objects.

The first server is called the *EmigrationServer*, it is responsible for sending outbound Ambassadors (*visit* and *migrate* operations), and for receiving inbound Ambassadors when they return from a *visit*. The second server is called the *ImmigrationServer*. This server is responsible for receiving incoming Ambassadors, invoking the appropriate method, and, if necessary, sending the reply to a *visit* operation when this invocation returns. The two servers on a given node are aggregated into an object termed an *Embassy*.

Both of these classes extend *UnicastRemoteObject*. The RMI system generates proxy classes for each and communication occurs, via the proxies, using RMI. Proxy objects for remote *EmigrationServers* and *ImmigrationServers* are obtained using the standard name server supplied with the RMI system.

To support the entrenched mode of operation, there is an interface *Entrenchable*, which is a subtype of the RMI *Remote* interface. For convenience there is a class *EntrenchableUnicastRemoteObject* which implements the *Entrenchable* interface and which extends the RMI *UnicastRemoteObject* class. A programmer has two alternative approaches to designate objects as entrenchable:

- Define a class extending *EntrenchableUnicastRemoteObject*. Then create an object of the newly defined class with a constructor calling the constructor of the *EntrenchableUnicastRemoteObject* class with a parameter indicating that the object should marked as entrenched.
- Define a class implementing the *Entrenchable* interface. Create an object of this class with a parameter that calls the class method, passing the new object as parameter. The programmer must also ensure that the new class implements the *equals* and *hashCode* methods to operate on the remote stub of objects of the class. This alternative is much more complicated but allows the new class to extend some class other than *EntrenchableUnicastRemoteObject*.

If neither of these approaches is use on a given object, then that object will always be accessed via RMI, even by Ambassadors in the same virtual machine.

8.5.2 The *visit* Sequence

The implementation of a *visit* operation is shown in Figure 82. It executes eight steps, as follows:

1. The client calls the *visit* method of the local *EmigrationServer*.
2. The *EmigrationServer* constructs a message containing a copy of the Ambassador, an indication of the method to invoke, the parameters to pass, and a unique tag that will be used

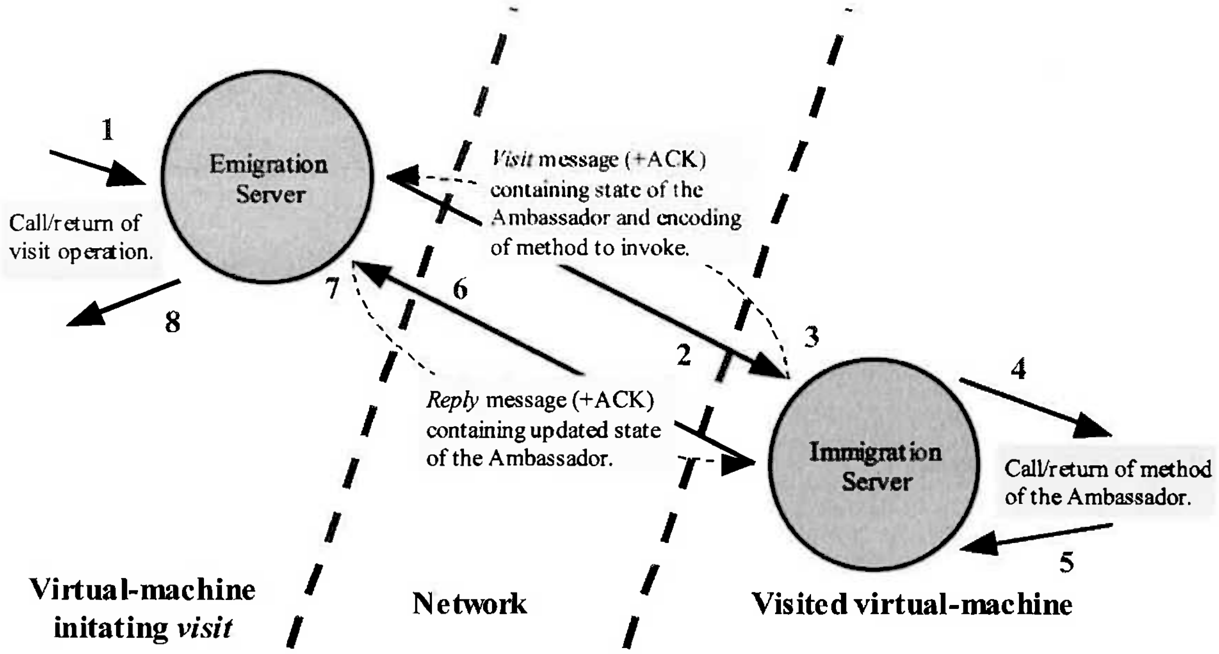

Figure 82 – Implementation of Visit.

to identify the reply to this visit. This message is sent to the remote *ImmigrationServer* via a method call to its local proxy. The calling thread is then put to sleep.

3. The remote *ImmigrationServer* receives the RMI request, checks the validity of the invocation, and creates a thread to execute it. The RMI then returns immediately, note that the reply to this RMI is not on the critical path for the interaction. Prior to the call, any remote references to local entrenchable objects in the closure of the call are replaced with references to the corresponding local object.
4. The invocation executes on the remote node.
5. The invocation returns, upon return, the *reply* method of the remote *ImmigrationServer* is called (this is a local call on the remote node), supplying the updated Ambassador object as parameter. Any (local) references in the closure of the Ambassador's instance variables to local objects implementing the *Remote* interface are replaced with remote references.
6. The *ImmigrationServer* calls the *receive_reply* method of the proxy for the *EmigrationServer* on the original node. This delivers the updated Ambassador, with the tag identifying the visit being replied to.
7. The *EmigrationServer* receives the reply as an RMI request. It uses the tag to locate and then overwrite the local copy of the Ambassador (the anchor), and wakes up the thread that called *visit*. The *EmigrationServer* sends a reply to the request, note; once again, this is not on the critical path of the interaction.
8. The thread returns from *visit* to continue execute client code.

Currently, the Ambassador class includes a stack of (tag, node) pairs in order to support nested visits. This is actually inefficient. Instead, the "live" copy of the Ambassador need include only the (tag, node) pair for the current *visit*. The pair for the next most closely enclosing *visit* can be stored at the anchor for the current *visit*, and so on. The implementation of a *migrate* operation is a subset of that described above for *visit*.

8.5.3 Concurrency

It has been seen in Chapter Four (Sections 4.1.5.1 and 4.1.7.2) that the introduction of concurrency leads to complications in the semantics of pre-conditions and post-conditions (and therefore of the notion of contract). These complications will arise similarly with Ambassadors, if multiple Ambassadors are permitted to be running concurrently at a given node. However, the vast majority of time in a typical Ambassador tour is spent in transit, rather than executing, so it may be reasonable to prohibit Ambassadors concurrency such that only one Ambassador is running at a given node at any time. A major advantage of this approach would be that the ordinary (and simpler) sequential semantics for pre-conditions, post conditions and contract can be retained, whilst still permitting concurrency within the system as a whole.

8.5.4 Managing Failures

As indicated previously, the management of failures in Ambassador tours is modelled on that for at-most-once RPC; essentially, Ambassador visits are treated as remote procedure calls. Failure detection for at-most-once RPC using a single time-out is subject to a trade-off between two aims:

- *Detecting genuine failures as early as possible* – requiring the time-out to be as short as possible.
- *Avoiding misdiagnosing delayed replies as failures* – for example, the remote computation might simply take a long time, and this ought not to be taken as failure – this requires that the time-out be as large as possible.

In at-most-once RPC, the solution is for the client to wait for the expiration of a short initial time-out, and if the reply has not been received, send an additional "probe" message to the server, setting a new time-out. The server, receiving the second message, should respond, either with the result of the RPC or with a "keep-alive" message, indicating to the client that the server is still processing the request. This process can continue, with the client setting increasingly larger time-outs (to avoid consuming excessive network bandwidth), until either the reply is received, or the server fails to respond to one of the probe messages, thereby indicating a possible failure. A general alternative to probe/keep-alive pairs is to have the server spontaneously generate keep-alive messages periodically for long running (remotely invoked) procedure calls.

The detection of failures in the Ambassadors system can be implemented using the techniques employed for failure detection in at-most-once RPC applied to *visit* operations. The main complication is that because of *migrate* operations, an Ambassador visiting a node may migrate from that node before returning to the visit origin. This means that a probe message may need to be forwarded round the network, which is expensive [Fowler 1986]. It might be better to employ the spontaneous keep-alive approach for Ambassadors that employ *migrate* operations in addition to *visit* operations.

8.5.5 Security

As noted in the discussion of semantics, the Ambassadors system raises two kinds of security issues:

- Verification of Ambassador code.
- Delegation of the client's privileges to the Ambassador.

These are discussed in the sections below.

8.5.5.1 Verification of Ambassador Code

The system can ensure that an Ambassador executes only the code that is intended by the client as follows:

- Require that the client use a private key to digitally sign the code of the Ambassador prior to sending it out, and send the signature along with the Ambassador on its tour.

- At each stage of the tour, prior to execution of the method for that stage, use the public key of the client to verify that the code of the Ambassador matches the signature sent with it. This public key is retrieved from a trusted third party.

This is a straightforward adaptation of the JDK 1.1 mechanism for code signing and is directly supported in JDK 1.2/Java 2 [Gong 1998].

8.5.5.2 Delegation of Clients' Privileges

In contrast to verification of Ambassador code, delegation of the security environment of the client to an Ambassador is challenging. Delegation of a principal's privileges is an issue of general interest to the Java community. Currently, it is assumed that each Java virtual machine is running on behalf of a single principal, as noted in the Java 2/JDK 1.2 security specification [Gong 1998]:

> *"Today the notion of a principal (eg., user) is implicit because each JVM is owned by one user. In the future, there will be a need to extend the existing concept of ProtectionDomain to include the notion of "running-on-behalf of" a principal.*
> *Therefore, we are actively looking to provide the following features in the near future:*
> - *explicit principal concept and classes*
> - *user authentication primitives (both password-based and otherwise)*
> - *cross-protection-domain principal authentication protocols*
> - *general mechanisms for authorization and delegation."*

A development of the core Java platform along these lines would provide the security functionality required by Ambassadors.

8.6 Implementations

Hollfelder [Hollfelder 1998], has refined the detailed Ambassadors design (without security) presented in the previous section and produced a number of prototype implementations. These include the following:

- An implementation of Ambassadors using RMI as the transport mechanism and the Java reflection interface to invoke methods on Ambassadors.
- An implementation of Ambassadors using TCP/IP to transport serialised Ambassador objects and the Java reflection interface to invoke methods on Ambassadors, with each Ambassador invocation executed by a separate thread.
- A single threaded implementation of the previous implementation, in which a single persistent thread executes all Ambassador invocations for a given virtual machine.

These prototypes have been significantly refined by the author of this thesis; the results presented in the next section are for these refined implementations and supersede previously published results (such as those in [Hollfelder, Detmold and Oudshoorn 1999]). These prototypes do not incorporate any security functionality.

8.7 Performance Results

The system was tested using the interaction described in Section 8.2. The communication mechanisms were compared based on wall-clock times for the interaction, adjusted for time spent in computation that is independent of the communication mechanism. Communication was over the public Internet, so to minimise the effect of variations in network performance, the experiment was conducted as follows:

```
for sample in 1..30 loop
  for numPrinters in 1,2,4,8,16,32 loop
     foreach communication-mechanism loop
        take-sample(numPrinters,communication-mechanism)
     end
  end
end
```

This uses an interleaving approach to attempt to distribute the effects of variations in the network evenly across sample for different mechanisms and different number of printers.

In the first experiment, the three nodes in the interaction were at The University of Adelaide (head office), The Australian National University (client) and the University of Southern California (remote department). Individual null RMI calls from the client to the remote department take 450 milliseconds and from the client to the head office take 36 milliseconds. Round-trip times for the ICMP ECHO_REQUEST/ECHO_RESPONSE (PING) protocol average 420 milliseconds between client and remote department, 33 milliseconds between client and head office and 430 milliseconds between head office and remote department.

If RMI is used to implement this interaction, then for p printers there will be $2p + 2$ RMI calls from the client to the remote department and one call from the client to the head office. Using the times given above, the minimum aggregate latency using RMI is $900p + 936$ milliseconds for p printers.

Many of these RMI calls can execute concurrently. In particular, all iterations of the first loop in the interaction can execute concurrently, with each thread executing the sequence of two calls contained in an iteration. The two calls prior to the loop can also execute concurrently. This leads to a critical path including three sequential calls to the remote department and one to the head office. The minimum time taken with this approach is 1386 milliseconds.

In the ideal case, information flows from the client to the remote department, from there to the head office and then back to the client. With the PING round trip times given previously, the minimum time spent transmitting data on the critical path is $210 + 215 + 16.5 = 441.5$ milliseconds.

Figure 83 shows the mean results for RMI, concurrent RMI, Ambassadors over RMI and the two versions of Ambassadors over TCP. The minimal cases for RMI, concurrent RMI and the ideal case are also shown. All versions of Ambassadors used the entrenched semantics rather than the hosted semantics. In all cases, the class loader caches at the remote nodes contained the Ambassador's code.

Inspection of the results for RMI reveals, as expected, that performance is linear in the number of printers. In contrast, for the Ambassadors implementation, the slope of the least squares interpolation is close to zero – the performance is nearly independent of the number of printers. At the top end of the scale, with 32 printers, iterative Ambassadors over TCP[106] resulted in wall clock times with a mean of 3118 milliseconds, versus 40434 milliseconds for RMI, giving a speedup of 13 times. 3118 milliseconds is still an order of magnitude worse than the minimum of 441.5 milliseconds required to support the necessary flow of information across the network. Nevertheless, the results for this implementation are promising, as it has not been optimised in any way.

The results for concurrent RMI are interesting. In principal, the time taken for the interaction using this technique is almost independent of the number of printers. In practice, this is not true, and the aggregate latency for concurrent RMI is linear in the number of printers, although with lesser slope than sequential RMI. It is likely that this is due to deficiencies in the implementation of multi-threading in Java (JDK 1.1 on Solaris), and that correction of these deficiencies will give constant time performance for this approach. At the top end of the scale, concurrent RMI averages 17390 milliseconds, consequently Ambassadors gives a speedup of 5.6 times.

[106] The graph shows the interpolation for interactive Ambassadors over TCP as having (very slightly) negative slope. This is not significant and should be ignored

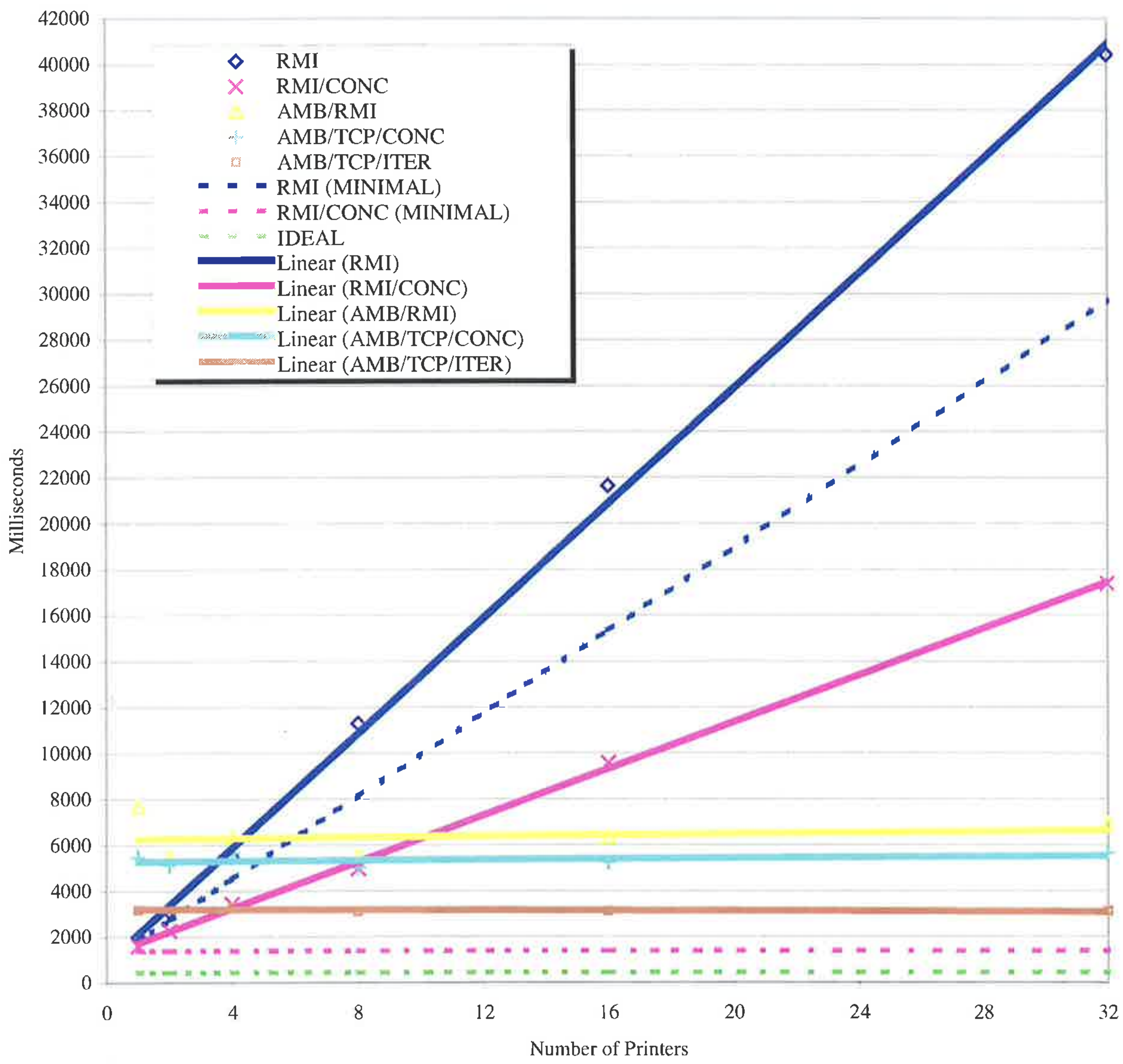

Figure 83 – Mean Wall-clock Times for a Worldwide Interaction.

The results for the other Ambassadors implementations are of the same order as the iterative TCP implementation. The use of multi-threading (using user level threads rather than threads supported by the operating system) in the TCP implementation results in a performance deterioration of a few percent. Using RMI as the message transport involves extra dispatch overhead for the EMI calls used to transport the data, and may involve multi-threading within the RMI mechanism. However, RMI may cache and re-use underlying network connexions, avoiding the need to establish a new connexion each time. The net effect of using RMI as the message transport is a further deterioration of a few percent.

Figure 84 shows results for an experiment with the same interaction conducted on three machines that are connected through an ATM network exhibiting sub-millisecond round-trips for the ICMP ECHO_REQUEST/ECHO_RESPONSE (PING) protocol. As before, the Ambassadors implementations outperform both RMI and concurrent RMI for large number of printers (and operations). As would be expected in a lower latency environment, the crossovers occur at higher number of printers than in the worldwide experiment.

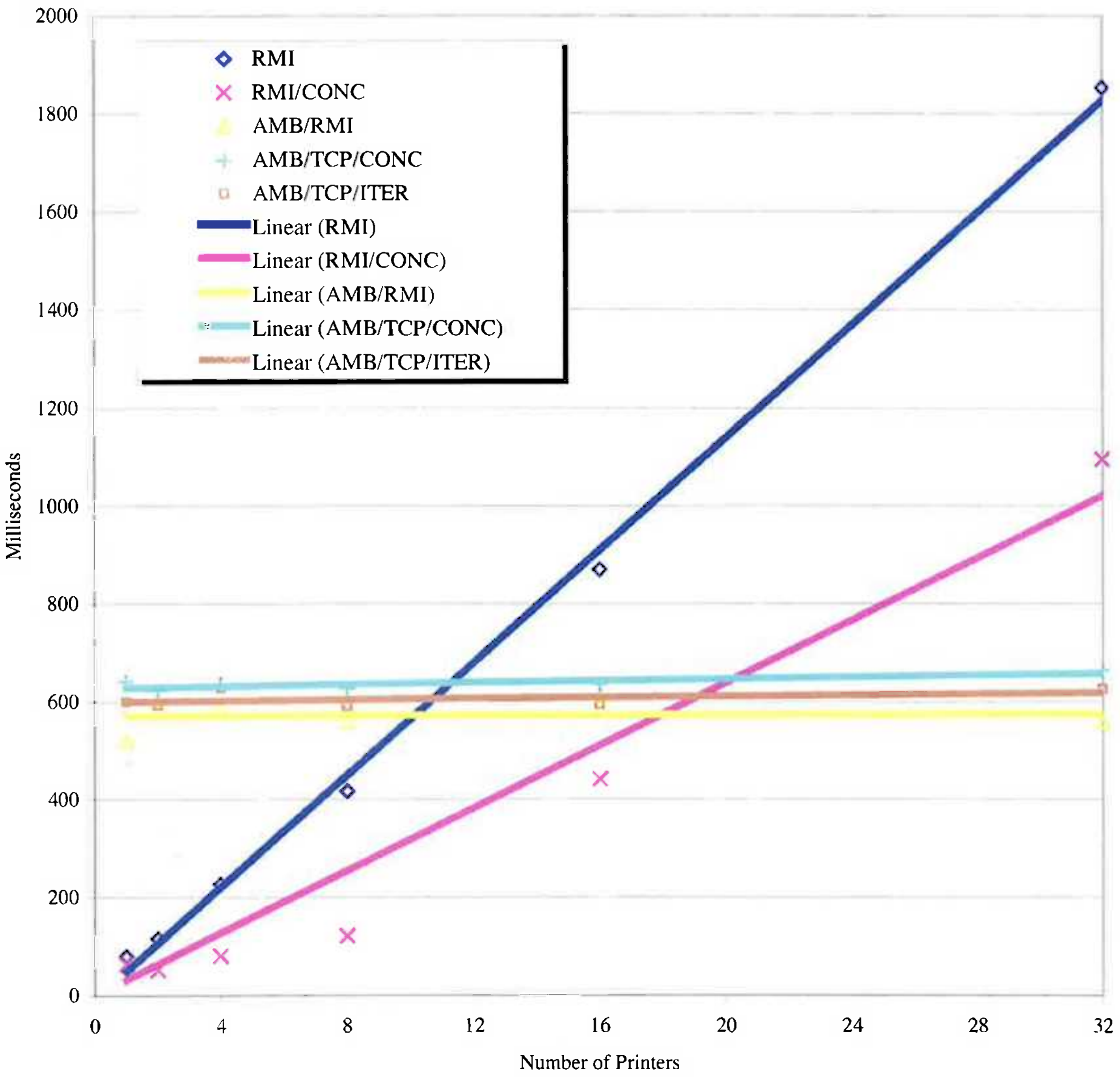

Figure 84 – Mean Wall-clock Times for a Local Interaction.

One interesting result shown in Figure 84 is that the implementation of Ambassadors over RMI outperform the TCP implementations in the relatively low latency environment of local area network. This may be due to RMI being tuned for use in such environments.

To examine the effect of entrenching Ambassadors, the local interaction was re-run with the following mechanisms:

- RMI (as before).
- Iterative TCP Ambassadors with an entrenchable resource directory object (as before).
- Iterative TCP Ambassadors hosted in the same virtual machine as a (non-entrenchable) resource directory object.

The results are shown in Figure 85 and clearly indicate the performance advantages of the entrenched mode. In fact, these results indicate that the current implementations of Ambassadors running in hosted (non-entrenched) mode deliver worse performance that does RMI when the interaction is run on the local network. If running on a worldwide network, the hosted version of Ambassadors performs better than RMI for a sufficiently large number of printers. However, the performance is still

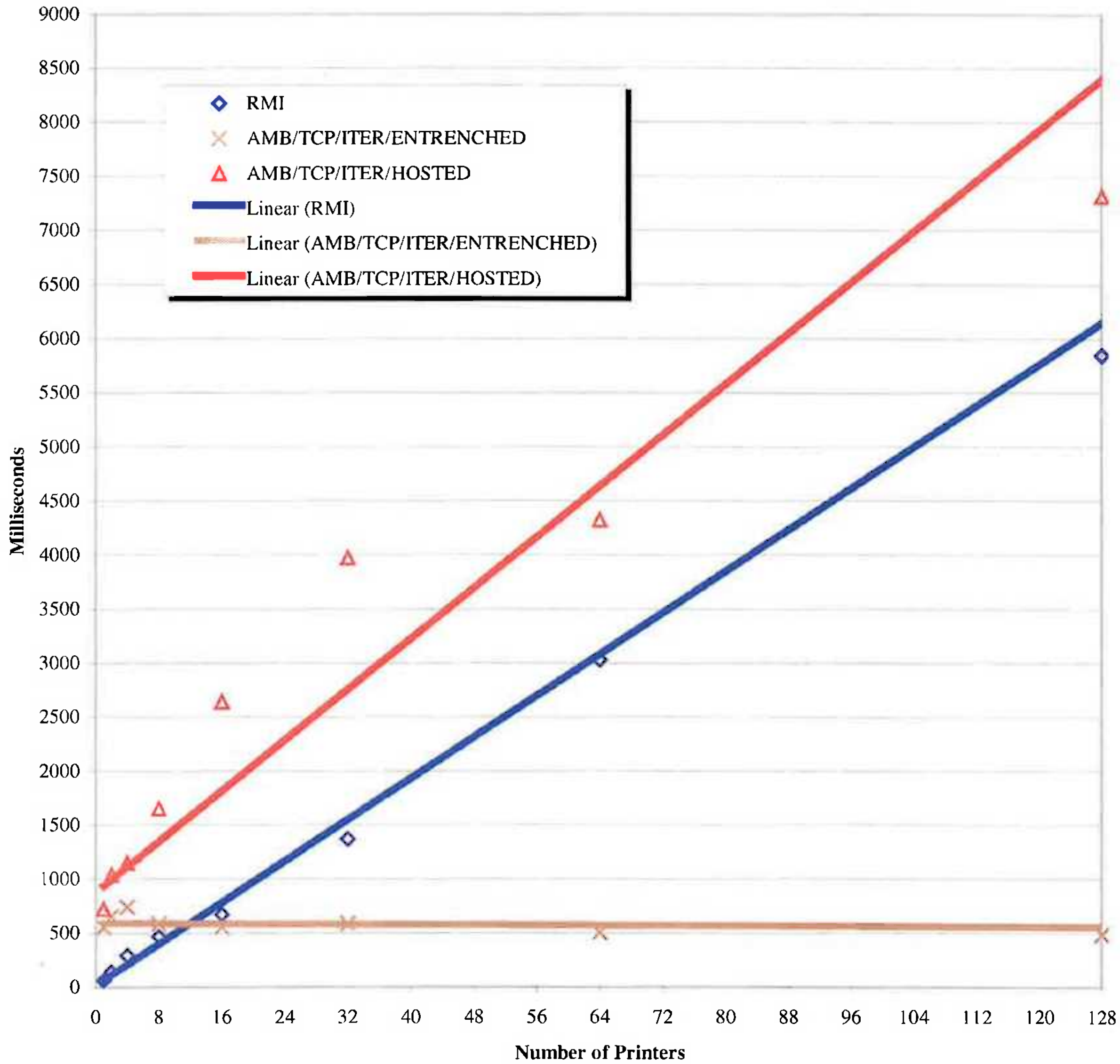

Figure 85 – Mean Wall-clock Times for RMI and for Entrenched and Hosted Ambassadors.

significantly worse than that of the entrenched mode and the least squares linear interpolation for the hosted version has a significant slope.

8.8 Optimisation

The Ambassadors implementations discussed above are completely unoptimised. Furthermore, the performance results indicate substantial room for improvement so optimisation is likely to be productive.

8.8.1 Reducing the TCP Connexion Overhead

The most obvious area for optimisation is in the use of TCP. All of the prototype Ambassadors implementations establish a new connexion each time an Ambassador is transmitted. In order to establish a connexion, TCP requires a round trip to be completed prior to transmitting any data on that connexion. Consequently, there is an extra round-trip in each step of the critical path for the Ambassadors implementation of the interaction. Using the PING roundtrip times, this equates to 420 + 33 + 430 = 883 milliseconds minimal overhead in the world-wide case, so eliminating these extra round trips would improve performance significantly.

Two approaches to remove these round trips from the critical path of Ambassador tours are as follows:

- Replace TCP, either with a connexionless protocol, or with a connexion-oriented protocol that can transmit data concurrently with connexion establishment (in anticipation that the connexion will be established successfully). This removes all unnecessary round trips from the critical path.
- Cache TCP connexions within the Ambassadors system, avoiding the extra round-trip for all but the first use of a given link.

The second approach is probably simpler, but, like all caching approaches, depends for its effectiveness on locality properties that may not be present in real systems. Further experimentation would be necessary to determine the choice to adopt.

To evaluate the performance of the caching option assuming good locality, a connexion caching implementation of Ambassadors over TCP with iterative servers was produced. This simply keeps the last used connexion open until a connexion to a different host is required. The structure of the example interaction is such that the connexions are established in the first iteration of the experiment and remain open throughout – *i.e.* the cache operates perfectly.

The results for the worldwide interaction using this implementation are shown in Figure 86. The mean aggregate latencies using this implementation are about half those for the equivalent non-caching implementation (a saving of about 1.5 seconds). These results demonstrate that connexion caching is a good strategy if significant locality is present. However, even the caching implementation is almost 300% worse than the ideal, indicating that there is still substantial scope for performance tuning.

8.8.2 Hosted Semantics Optimisation

The second area for optimisation is to improve the performance of RMI calls from hosted Ambassadors to server objects in the same virtual machine. The aim with the hosted version of Ambassadors is for calls to server object to have the same semantics as RMI calls, but with improved latency. The semantics distinguishing RMI calls from local calls are as follows:

- RMI calls may fail, raising an exception generated by the RMI mechanism.
- Parameters and return values are passed by reference in local calls but are passed by value (implemented by closure copy for non-remote objects) in RMI calls.

Notice that a call mechanism in which calls never fail is still conformant to these semantics.

The closure copy operations in an RMI call are actually implicit. Consider the RMI call sequence, as follows:

1. Call stub method.
2. Serialise call.
3. Transmit serialised call over network to server.
4. Deserialise call.
5. Dispatch call.
6. Execute method.
7. Serialise result.
8. Transmit serialised result over network to client.
9. Deserialise result.
10. Return to caller.

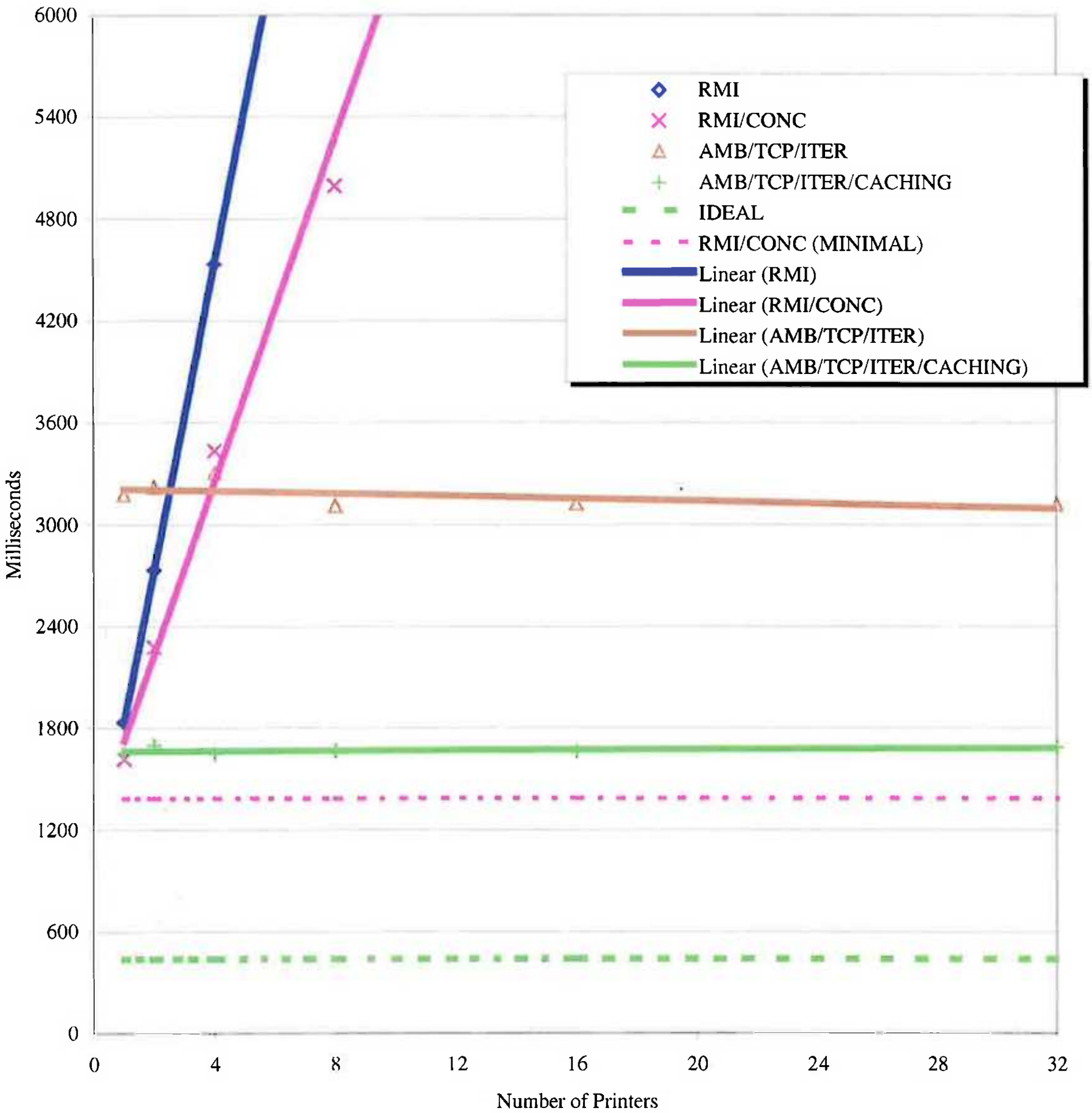

Figure 86 – Evaluating the Effect of Connexion Caching in the Worldwide Interaction.

In this call sequence, closure copy of the parameters is a side effect of steps 2 and 4, and closure copy of the result is a side effect of steps 7 and 9.

Now, it is possible to provide the closure copy semantics of RMI for calls to objects in the same virtual machine but to avoid most of the over-head of RMI; consider the following:

1. Call stub method.
2. Clone parameters.
3. Execute method with cloned parameters.
4. Clone result of method.
5. Return cloned result.

This call sequence is indistinguishable from RMI since, as mentioned previously, the semantics allow calls to fail, it does not require that calls actually fail.

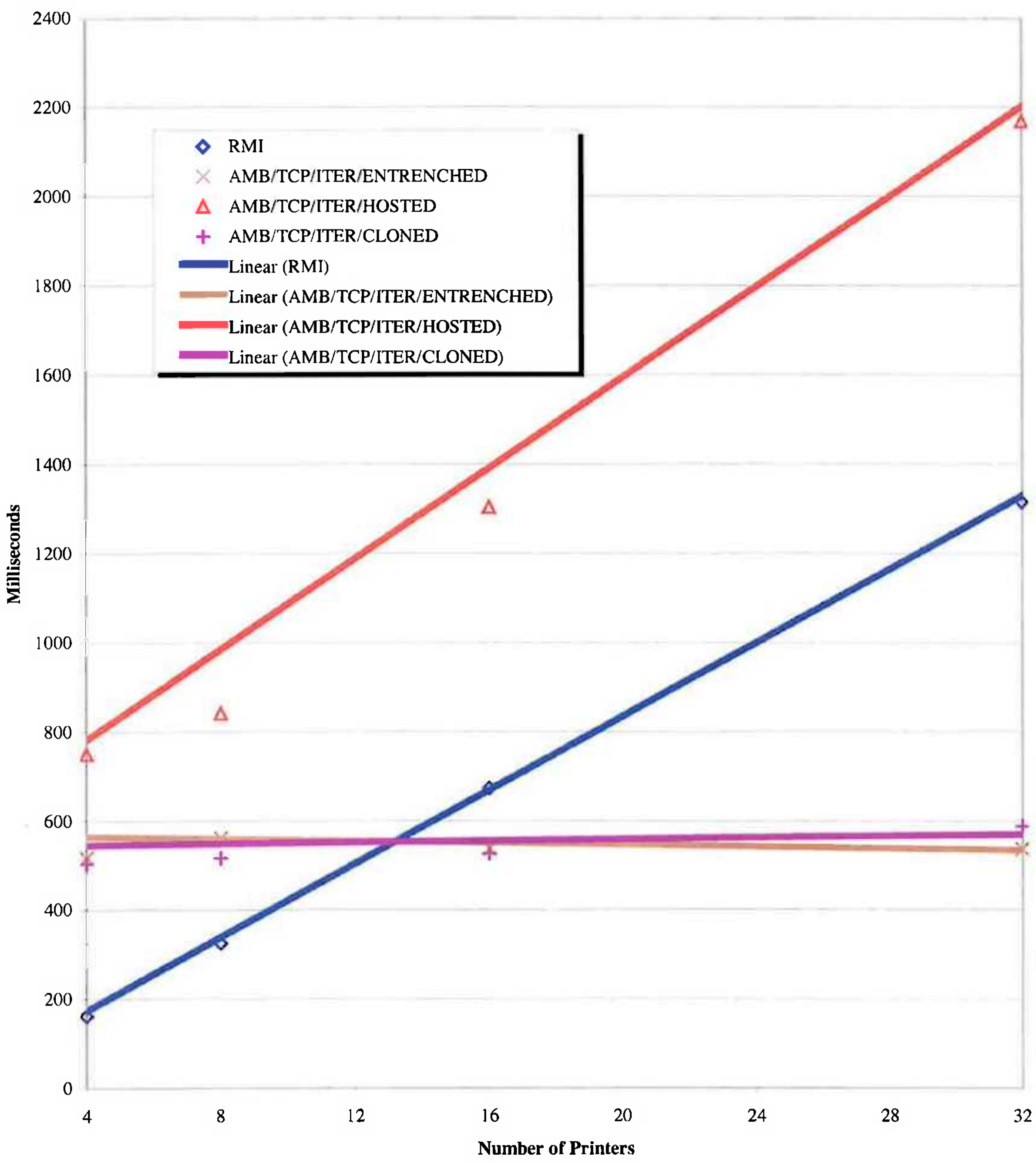

Figure 87 – Comparing the Cloning and Original Implementations of the Hosted Semantics.

Figure 87 shows results for an implementation of the hosted mode using the above call sequence instead of RMI. This improves performance for the example interaction to the point where it is essentially indistinguishable from the entrenched mode. The example interaction is not ideal for comparing the entrenched mode with the hosted (and cloned) mode, since the amount of closure copying in this example is minimal. However, it is fair to compare the cloning implementation of the hosted mode with the RMI implementation (denoted AMB/TCP/ITER/HOSTED in the figure) and cloning clearly has the better of this comparison.

The cloning implementation of the hosted mode is similar to that of the entrenched mode. Recall that in the entrenched mode local references are substituted for remote references to objects in the local virtual machine. In the cloning implementation of the hosted mode, remote references to object in the local virtual machine are substituted with references to stub implementing the cloning call sequence given previously.

Consequently, the cloning approach appears to be promising as a means to obtain adequate performance from the hosted mode. In the experiment reported in Figure 87, the stubs implementing the cloning call sequence where implemented manually. A stub compiler similar to that generating RMI stubs is needed in a production implementation of the system using this approach. Alternatively, it may be possible to implement cloning using reflection, avoiding the need for stubs entirely.

8.9 Middleware Activated

As demonstrated in the previous sections of this chapter, improved performance in wide area systems is one major reason to replace the current generation of passive middleware with active middleware. However, this is not the only reason. Some additional reasons to favour the active approach include the following:

- *Increased flexibility for legacy component adaptors* – One of the major roles played by the current generation of middleware is in the creation of adaptors that make legacy software systems available as services in a modern distributed object system. Passive middleware has limited flexibility in the description of legacy component adaptors because of its static nature. These limitations may result in a requirement to create code within the legacy system to act as an adaptor between the system and the middleware. Active middleware is more flexible in this regard, supporting the creation of adaptors capable of dealing with all sort of unusual variations in the means of interfacing with the legacy system. The active middleware approach is analogous to the human ambassador who is conversant with local customs, whereas the passive approach is like an attempt to impose a single protocol on the process of international diplomacy.
- *Improved support for disconnected operation* – The networks supporting distributed systems exhibits loss of connectivity, either because of system failure or by design (as a user's lap-top is unplugged, for example). Active middleware can continue to operate and make further progress in the presence of a loss of connectivity. In contrast, passive middleware can only, at best, suspend operations over the part of the network affected, then wait for connectivity to be re-established and then resume those operations.

The main advantages of the traditional approach are as follows:

- *Simpler programming model.* – That is, RPC/RMI as opposed to the special *visit* and *migrate* operations of the Ambassadors mechanism. In particular, RPC/RMI calls are statically type-checked, which his not the case with the current version of the Ambassadors mechanism. Overcoming this limitation is the subject of active further research.
- *Simpler resolution of security issues* – Security in RPC/RMI is relatively straightforward because the only autonomous actor on behalf of a principal is the principal itself (the client). For example, a client can digitally sign the requests it transmits, enabling their integrity to be verified. In contrast, Ambassadors are autonomous actors on behalf of a client. This raises a plethora of complex and interesting security issues. These would need to be resolved before active middleware can be used in secure environments.

The most likely candidate for future middleware would seem to be a hybrid of the active and passive approaches. A Java and Web-based version of this approach is illustrated in Figure 88, which shows a two tier system; and in Figure 89, which shows a three tier system. As can been seen in the figures, active middleware computations are physically located in the tier below the appropriate logical tier.

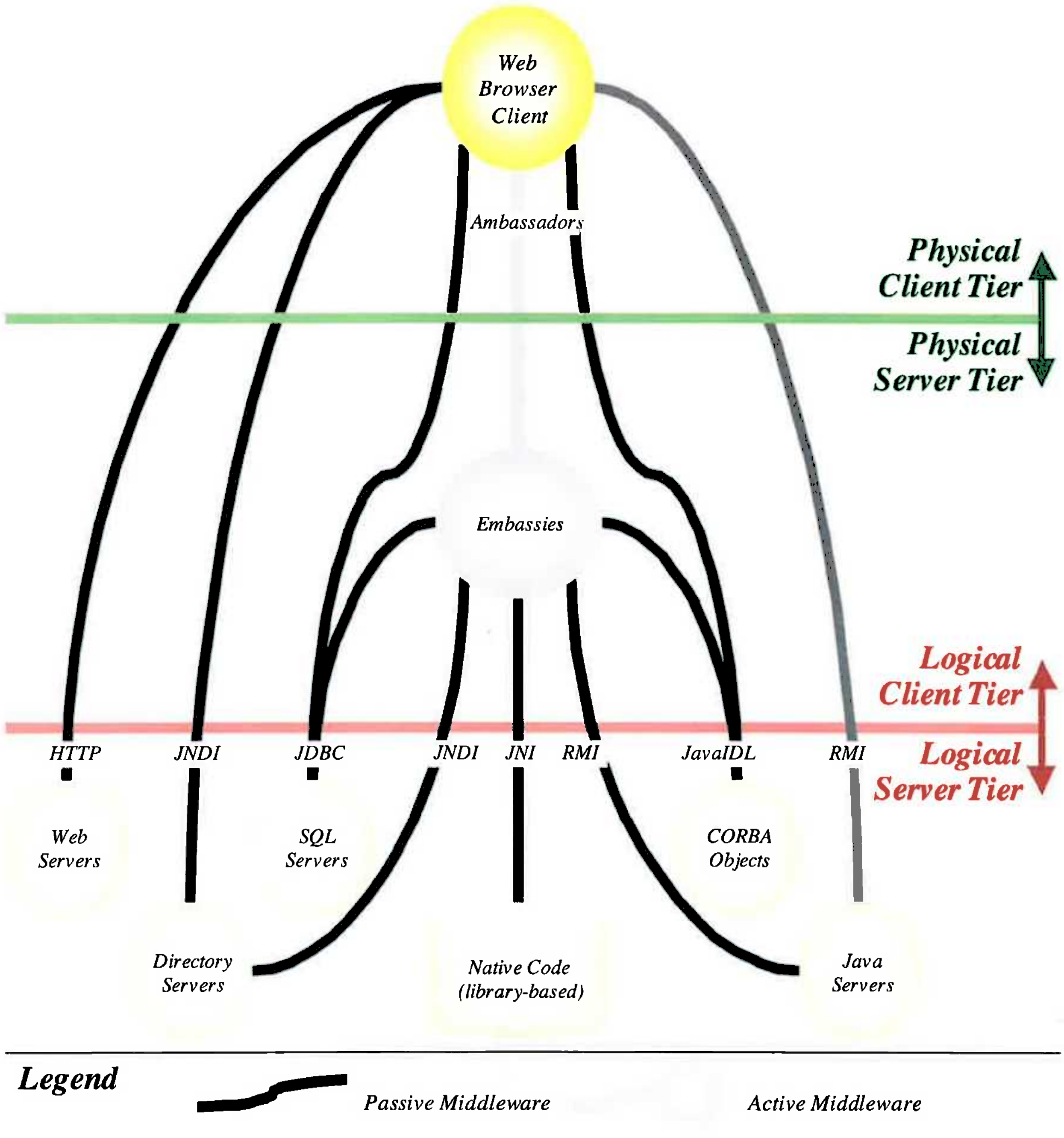

Figure 88 – Two Tier Client/Server System with Active and Passive Middleware.

8.10 Comparison

This section compares Ambassadors to two kinds of mechanisms:

- Mobile agent systems.
- Support for mobile objects in the Emerald and DOWL distributed object-based programming languages.

8.10.1 Mobile Agents

The ubiquitous mobile code facilities of Java have made it an excellent basis for the construction of mobile agent facilities [Chang and Lange 1996]. There is a superficial similarly between

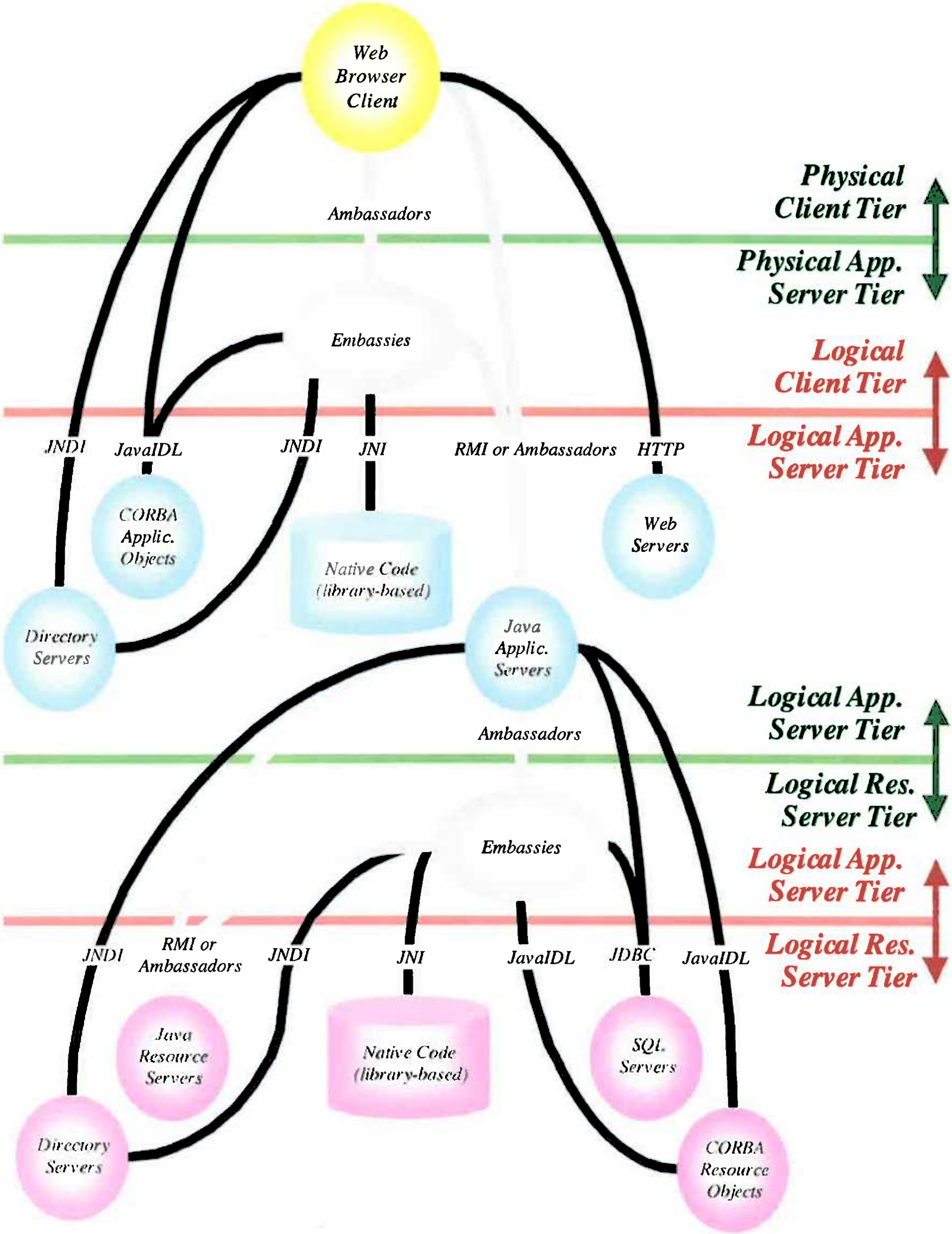

Figure 89 – Three Tier Client/Server System with Active and Passive Middleware.

Ambassadors and such mobile agent systems, but in fact, there are very considerable differences between the two.

- Ambassadors are purely a communication *mechanism* which can be used in whatever way is desired. In contrast, Mobile agent systems provide a specific set of *policies* – for mobile agent functionality – built on top of an internal communication mechanism. This mechanism is available independent of the (agent based programming) policy, thereby limiting application of the mechanism to situations where the policy is appropriate.

- Ambassadors are purely concerned with communication, any computation carried out by an Ambassador is only to coordinate and control that communication. In contrast, mobile agents are a vehicle for computation.
- Ambassadors are designed as a computation mechanism for distributed systems and deal with the failures that may occurs in such system with at-most-once semantics. In contrast, mobile agents do not define any particular semantics when deployed on systems having components that may fail.
- An Ambassador operates as directed by its originating client. In contrast, mobile agents typically have more freedom to pursue a separate agenda.
- Ambassadors migrate between nodes at the point of method calls, thus providing a structure to the distributed computation using the standard mechanism for program structure (procedure call). Mobile agents typically define structure in an orthogonal manner.

8.10.2 Emerald and DOWL

Emerald [Black *et. al.* 1986] provides extensive support for object migration, similar facilities are provided in the DOWL system [Achauer 1993]. However, as is shown below, the Ambassadors system is completely different in goals and direction, although the Emerald/DOWL object migration facilities could be used as the basis of a version of Ambassadors.

There are two Emerald/DOWL facilities for object migration. The first is call-by-move, the second is explicit assignment to the location property of an object.

Call-by-move is a mechanism for remote procedure calls or remote method invocations. Given an invocation of the form:

```
server.method(parameters)
```

In this invocation, *server* denotes an object on a remote node, the *parameters* of the invocation of *method* are moved to the node containing *server*. The execution of a method can then access these parameter objects locally (on the node containing the target). The idea here is to improve efficiency of execution of a method defined by the server class, by preemptively making local the objects needed in the execution of that method.

With the Ambassadors system, an Ambassador and the parameters of a method call thereon are moved to the location of a third party (a server), in order to improve the efficiency of a method defined by the Ambassador. This method would execute more than one call on the server: the efficiency improvement is obtained by avoiding separate network round-trips for each such call. The essential point is that the method is defined by the client programmer (who creates the Ambassador class) rather than by the programmer of the server.

One could use call-by-move to provide the semantics of the visit operation by having each server class define a method as follows:

```
void call_action(Ambassador a,Params p)
  { a.action(p); }
```

A client calling this method with call-by-move semantics would achieve the semantics of visit in that the Ambassador would first be migrated to the location of the server, and then have the call to the *action* method executed upon it. There are, however, some problems with this approach:

- It requires the implementor of each "server" class to implement or inherit the *call_action* method, whereas Ambassadors do not require changes to server classes.
- There is no analogous case for the *migrate* operation, only *visit* is supported.

- The client cannot specify which method should be invoked on the Ambassador, it is always the *action* method. Furthermore, the implementor of the *action* method must type-check the parameters. With Ambassadors, method selection and type checking are provided by the Java reflection API.

In summary, the call-by-move mechanism is not entirely suitable for applications for which Ambassadors are intended.

The second Emerald/DOWL mechanism is the ability to migrate objects explicitly. Each object (in DOWL[107]) has a special property, *$LOCATION* and assignment to this property causes the object to migrate to the assigned location. Using this facility, it is possible for the implementor of an Ambassador to obtain the semantics of the visit operation. The idea is that each method defined by an Ambassador class has the following form:

```
void action1(Server on_server, … parameters ) {
  saved_location = this.$LOCATION;
  this.$LOCATION = on_server.$LOCATION;
  … body of method, calls methods of on_server …
  this.$LOCATION = saved_location;
}
```

There might be several such methods for a given Ambassador class.

This approach overcomes many of the limitations of the previous (call-by-move) approach. There are still some problems however:

- In general, the ability to assign to the *$LOCATION* property requires migration of any threads engaged in operations on the object[108]. Consider a case where a method is called at location *L*, for an object *O*, and that method assigns location *M* to the *$LOCATION* property of *O*. When that method call returns, execution will continue at location *M*, hence the entirety of the execution stack will have to have been migrated along with *O*. In contrast, the Ambassadors system guarantees that migrations only occur at procedure call boundaries, and that return from a *visit* (including return of a method invoked with the *migrate* operation) will migrate the Ambassador back to the node that initiated the *visit*. Consequently, the Ambassadors system does not require thread migration: it is enough to suspend a thread calling a method that initiates a *visit* and resume it when the *visit* returns.
- There are two problems with thread migration. First, it is more expensive, in that the closure of the thread's execution stack must be migrated, rather than the more limited closures in the Ambassadors system. Secondly, thread migration is complicated in cases where language level threads have operating system support (as must be the case to take full advantage of threads), and there is not also operating system support for thread migration.
- Although it is possible to extend the programming convention adopted for *visit* to support *migrate*, the extension is not straightforward, nor is it easy to follow.

In summary, whilst it is possible to simulate the semantics of Ambassadors using the Emerald/DOWL facilities, there is additional cost in performance and it is significantly less convenient for programmers.

[107] The Emerald mechanism differs only slightly from its DOWL derivative.

[108] If the programmer follows the convention presented in the *action1* method, there is strictly no requirement for thread migration, however, there is no way to ensure that programmers always follow this convention, therefore the Emerald system must always migrate threads.

Capability	**Ambassadors**
Replies	True
Early Reply	False
Reply Matching	Full support (3/3)
Reply Synchronisation	Implicit (3/3)
Reply Kinds	All
Coordinated Termination	Automatic (3/3)
Syntactic Disruption	Significant (1/3)
Semantic Disruption	Consistent (3/3)
One-way Requests	False
Request Forwarding	Linguistic (3/3)
Peer Symmetry	False
Reply Entailment	Simple Client/Server (2/3)
Reply Races	No support (1/3)
Multiple Replies	False
Interaction Structure	Fixed (1/4)
Linguistic Heterogeneity	True

Capability	**Ambassadors**
Type Checking Timing	Dynamic (1/3)
Type Checking Clustering	Compatible (2/2)
Assertion Coverage	Not Applicable
Intrinsic Assertion Specification	
Inseparable Assertion Specification	
Declarative Assertion Specification	
Early Assertion Checking	
Precondition Synchronisation	Not necessary – Ambassador running on server node.
Asynchronous Contract Completion	
Reply Specification Coverage	Simple (2/3)
Multiple Reply Entailment	False
Interaction Structure	Fixed (1/4)
Intrinsic Reply Specification	True
Inseparable Reply Specification	
Declarative Reply Specification	
Positive Reply Specification	
Reply Sending Enforcement	
Illegal Reply Handling	Prevention (4/4)
Illegal Reply Rejection Timing	Not Applicable
Extra Reply Handling	Prevention (4/4)
Extra Reply Rejection Timing	Not Applicable
Asynchronous Method Call Exception Handling	Not necessary – Ambassador running on server node.
Exception Handling Contexts	Not Applicable.
Exception Handling Flexibility	

Interaction	**Ambassadors**	**Caching Ambassadors**
$I_{SSS}(f,r,o,n,m)$	$6f$	$2f$
$I_{SSM}(f,r,o,n,m,s,r',o')$	$3(s+f)$	$s+f$
$I_{AD}(a)$	$3(a+2)$	$a+2$
$I_{COSS}(f,r,o,i,n,m)$	$6f$	$2f$
$I_{COSM}(f,r,o,i,n,m,s,r',o',i',ir')$	$3(s+f)$	$s+f$

Overlapped Computation	(1/6)
Request Transmission Policy	Eager (3/3)
Reply Transmission Policy	

Table 24 – Assessment Checklist – Ambassadors.

8.11 Assessment of Ambassadors

The performance of Ambassadors has been assessed empirically above. This section assesses the mechanism analytically according to the criteria identified in the framework of Chapter Five, the assessment is summarised in Table 24.

Ambassadors incorporate the notion of reply in the form of return visit operations. However, there is no facility for early reply. As with RPC/RMI, with Ambassadors, reply matching and synchronisation are implicit. Replies of all kinds can be supported, since the instance variables of an Ambassador can be declared to be of any type desired.

The structural constraints on Ambassador tours ensure that Ambassadors normally achieve coordinated termination automatically. The main exception to this rule is the case where an Ambassador creates a thread on a node that it is visiting and that thread continues after the Ambassadors has left. This is probably undesirable even without the negative implication for coordinated termination, and could be avoided by prohibiting Ambassadors from starting threads or by terminating any such threads when the Ambassador departs.

Ambassadors cause significant syntactic disruption of existing sequential code, which will have to be rewritten to take advantage of the mechanism. Surprisingly perhaps, Ambassadors cause minimal semantic disruption of existing sequential code when run in with the entrenched semantics. This is because calls from such Ambassadors are local method calls, with familiar semantics, as opposed to remote calls.

As a result of the *migrate* operation, Ambassadors support request forwarding (more appropriately termed *visit* forwarding in this context). Using *visit* forwarding, it is possible to create administrators that forward Ambassador *visits* to achieve load balancing and similar requirements.

Ambassadors are intended for use in a client/server context. No work has been done on possible adaptation of Ambassadors to the peer context. As such, reply entailment for Ambassadors follows the simple client/server model, with a single reply for every request (*visit*) and therefore no opportunity for races between replies.

Ambassadors are used as active middleware, with all the facilities of the ubiquitous Java language available for interfacing with computational components of the system, including components written in other programming languages. Consequently, Ambassadors can be used to construct heterogeneous systems.

Ambassadors currently perform type checking dynamically. Addressing this flaw is the subject of future research. The dynamic type checks in the current system are clustered at points where the Ambassador arrives at each node on its tour.

Ambassadors, like Java, do not provide any intrinsic support for assertion checking, although such could easily be added. Non-concurrent Ambassadors avoid the difficult issues arising from the interaction between contract-based programming and concurrent clients. An Ambassador running on a node can be given exclusive access to the server on that node, this is a return to the easily understood sequential semantics. Similarly, asynchronous exception handling is relegated to a non-issue.

Transmission of replies is implicit from the point of view of users of the Ambassadors mechanism. Consequently, illegal and extra replies are prevented, as was the case with RPC/RMI.

Chapter Nine

Conclusion and Future Directions

Section 9.1 of this chapter summarises the contributions of this thesis. Many open issues remain, some issues of particular interest are discussed briefly in Section 9.2.

9.1 Conclusions

This thesis aims to contribute to the body of knowledge in the area of the construction of distributed object systems to execute efficiently on the worldwide scale. The major determinant of system efficiency on such a scale is the massive latency of individual network communication; responding to the challenge posed by this degree of latency has been the driving force behind this work..

Whilst it is necessary that worldwide systems be efficient, it is also important that this efficiency is not bought at the price of compromising the ability to apply modern software construction techniques in the development of said systems. Distributed systems need a high level programming model at least as much as do traditional centralised systems. The object-oriented programming model is becoming the dominant programming model in software engineering in general and it so happens that this model is uniquely suited to the construction of distributed software. Chapter Two examines the issues inherent in the construction of distributed systems. Chapter Three presents a variant of the object-oriented programming model. Chapter Four integrates the previous two chapters into a discussion of distributed object systems, the model used in the remainder of the thesis.

Chapter Five introduces the first major contribution of this thesis, providing a detailed and novel exposition of the issues facing programmers in the development of worldwide distributed object systems. This culminates in the development of a detailed framework for assessment of programming mechanisms for communication in such systems. This framework is applied to the assessment of a selection of existing communication mechanisms in Chapter Six.

Chapter Seven contains the second major contribution, the development of two related communication mechanisms, Entitlements and Responsibilities. These are designed to reduce the impact of latency in worldwide distributed systems without requiring any facility for code migration. The Entitlement mechanism is reasonable effective reducing the impact of latency in simple client/server systems whilst retaining a simple programming model. The Responsibilities mechanism performs less well and is less easy to use, but offers additional flexibility in the realm over complex interactions and interactions between processes running as peers. The two mechanism are related via subtyping. There is consequently significant integration of the two, and use can select the mechanism that best suits their needs on a case by case basis.

Chapter Eight describes the third and final major contribution of the thesis, the development of a third mechanism, Ambassadors, as a form of Active Middleware based on Java. Ambassadors use the ubiquitous code mobility offered by Java to minimise the effect of latency in worldwide distributed systems whilst providing an easy to use programming model. With Ambassadors, the goal of minimal latency for a wide class of interactions is realised in principle – there is nothing in the semantics of the construct that excludes attainment of that goal. The empirical performance results obtained from several prototype implementations of Ambassadors are promising, whilst still leaving room for improvement and scope for future work.

9.2 Future Directions

The main issues of interest for the future development of the research described in this thesis are as follows:

- An investigation into the incorporation of Mentat-like dependency graphs into the Responsibilities/Promises system. This would differ from Mentat in that the graphs would be computed on the fly, as requests are issued, rather than being determined at compile-time. Consequently, the requirement for modifications to implementation language compilers would be removed, enabling application of the middleware to be far more general.
- Improvement of the Ambassadors programming model, particularly in the area of static type safety.
- Integration of the Ambassadors facilities described in Chapter Eight with the Responsibilities/Entitlements system, or with the Mentat-like development of that system contemplated in the previous point. Since the Responsibilities/Entitlements system already includes a form of parallelism, the should also include development of Ambassadors in that area.
- A detailed examination of the security issues raised by Ambassadors in particular and by mobile code in general.

Each of these issues is now discussed.

9.2.1 On the fly Computation of Call Dependency Graphs

In the Entitlements mechanism, the responsible supplier for an entitlement provides the fulfilling value of that entitlement to third party nodes that need it in response to messages from those nodes. In contrast, in Mentat, the client calling the operation producing a result provides the server of that operation with a dependency graph indicating the nodes to which the result is to be sent. This graph is constructed by static analysis of the way in which the result is used in the client after the call.

The Mentat approach is slightly more efficient in terms of latency, in that there is no need for a round trip in order to obtain a result value on a third party. However, the implementation of this approach using a compiler (as Mentat does) is undesirable since there would need to be a compiler for every client programming language in the system.

An alternative is for the client to calculate dependency graph on the fly, by examination of the requests outgoing from the client. Each time a request containing a given unfulfilled entitlement is sent from the client to a third party that has not previously been sent the entitlement, the client sends a message to the responsible supplier. This message contains the identity of the third party node. The responsible supplier maintains a dependency graph of each entitlement it has responsibility for, adding nodes to the graph in response to messages (from the client, and from third parties that send the entitlement to other third parties).

This mechanism is quite similar to the current mechanism for third party demand registration. It differs in sending this registration as soon as the requirement is known. Consequently, it should match any latency reduction advantage Mentat has over the current mechanism.

9.2.2 The Ambassadors Programming Model

Ambassadors invocations are type safe, but type safety is enforced at run time rather than statically. This reduces programmer productivity, even in the development of small programs such as the example used to test the performance of the system, and will be a significant problem in large scale use of Ambassadors.

There is no fundamental reason why Ambassadors cannot be statically type checked. Consider the following *visit* operation:

```
EMIGRATION_SERVER.visit (server, amb, "m1",new Object[]{p1,p2 ...});
```

Type checking of the invocation resulting from this *visit* is as follows:

- Checking that the type of *amb* contains at least one candidate method *m1*, having void return type.
- Checking that the types of the actual parameters *p1*, *p2*, ... conform to the types of the formal parameters of (one of) the candidate method(s) *m1*.

This is done dynamically, by the *ImmigrationServer* for the node containing *server*, but all the information used in the type checking is available when the call to the visit operation is compiled, so it could instead be done statically.

Given that static type checking is feasible, how is the system to support it? There are two alternative approaches:

- Modify the compilation process to type check calls such as the above statically.
- Modify the mechanism so that Ambassador calls are subject to the standard static type-checking mechanism provided by Java.

The first of these is unreasonable, it complicates the compilation process by adding a second context in which type checking of calls occurs. Considerable continuing effort would be required simply to ensure that type-checking in this context operated in the same way as in the context of ordinary method calls.

The second is more promising [Detmold 2000]. The idea is that Ambassador operations appear as normal calls, such as the following:

```
amb.m1(server,p1,p2,...);
```

Syntactically, this is an ordinary method call, and is type-checked as such by the language compiler.

Semantically, the *amb* object in the client is a system-generated sub-class of the actual Ambassador class. This uses dynamic binding to redefine the *m1* method to execute a visit operation to the virtual machine containing server and call the original version of the *m1* method at that node. The node to visit is determined by some sort of programming convention, some possibilities include:

- Require that visitable objects (such as *server* above) implement an interface (*VISITABLE* say). The node to be visited is the node containing the first visitable object found in a left to right scan of the method parameters.
- Alternatively, consider all *VISITABLE* parameters and visit the node containing the largest number of such parameters.
- Alternatively, require that the first parameter to any method that is subject to implicit migration implement the *VISITABLE* interface, and visit the node containing the object passed as this parameter.

This idea provides a basis for future development, but considerable refinement of the semantics is still needed. In addition, some facility to obtain a *migrate* operation rather than a *visit* needs to be included.

9.2.3 Integration of Mechanisms

A aspect that is common to all three mechanism described in this thesis is that they are all related to method calls:

- Entitlements are created as the result of method calls.
- Responsibilities hold the result of method calls.

- Ambassadors are caused to migrate by method calls.
- Once migrated, Ambassadors use method calls to communicate with servers.

It may be desirable to integrate the mechanisms somewhat. In particular, Ambassadors could have entitlement semantics on the originating client. This would allow that client to continue after sending an Ambassador on a visit, blocking only when it executes a strict operation on that Ambassador.

The Entitlements and Responsibilities mechanisms provide a form of parallelism (between caller and callee), whereas an Ambassador tour is currently a sequential collection of steps. It may be profitable to add parallelism to the Ambassadors system, allowing some of those steps to occur concurrently. The main issues are as follows:

- A given Ambassador tour starts out as a single visit, so the originating client is expecting a single Ambassador to return to it, rather than an arbitrary number of copies of the Ambassador. Each introduction of parallelism to a tour creates a fork in the tour graph and extra copies of the Ambassador. In order to ensure that a single copy eventually returns, there should be a join operation corresponding to each fork.
- The join operation re-integrates several copies of an Ambassador into a single copy. Each of the pre-join copies may have a different state in its instance variables, these need somehow to be integrated to produce the state of the post-join Ambassador. It is likely that programmers will have to provide code to manage this re-integration.

Further down the track, Ambassadors, Entitlements and Responsibilities could be brought together with other communication constructs to form a unified model for object-based distributed programming.

9.2.4 Ambassadors and Security

A great deal more work needs to be done in relation to the security implications of mobile code in general and Ambassadors in particular.

Recall that the requirement is that an Ambassador executing on behalf of a given client has the privileges of that client (or some subset of those privileges determined by the client. Modern security mechanisms are based on public key cryptography, with which a client can digitally sign the remote operations it submits in order to establish its credentials to execute those operations. The critical aspect is that the ability to sign an object (such as a request) is contingent on knowledge of the signing principal's private key. The integrity of the mechanism is wholly dependent on the fact that a principal's private key is known only to the principal. In particular, the private key should never be transmitted over the network in the clear.

Now, suppose an Ambassador issues an operation that is restricted in security terms. In order to have that operation execute with the privileges of the client, it need to sign the operation with a private key identifying the client. Two obvious ways this might be done are as follows:

- Send the operation back to the client to be signed. This is secure[109], but defeats the purpose since it re-introduces round trips between client and server for each operation into the critical path, thereby destroying the latency reduction that motivated Ambassadors in the first place.
- Send the private key with the Ambassador, permitting it to sign operations locally. To avoid compromising security completely, this would need to be encrypted. The obvious choice is for the sender to encrypt with the receiver public key for each hop on the tour. The receiver can then decrypt the client's private key with its own private key before it start to execute the Ambassador invocation. The consequence of this is that the client's private key becomes known

[109] Notice that there can be no reliance on an ability to authenticate the Ambassador as the source of the signing request. Instead, the client should independently verify that the request is, in fact, as intended. This may itself be difficult.

to any node the Ambassador visits, and that these nodes can thenceforth impersonate the client. This is highly undesirable.

It is not at all clear that there is good solution to this problem. The interaction between mobile objects and security is one of several challenging issues encountered in undertaking this research.

Appendix A

Type Relationships for Responsibilities and Entitlements

Chapter Seven introduced some sub-typing relationships for responsibilities and entitlements, in that for a given type *T*, the type *RESP*[*T*] is a sub-type of the type *RESP*[*T*] which is also a sub-type of *T* if *T* is a purely indirectly accessed type. This sub-typing relationship is capable of being expressed in most object-oriented programming language with explicit sub-typing (that is sub-typing relationships that are declared rather than inferred) and supporting parametric polymorphism, with entitlement and responsibility as user defined parameterised classes. There are in principle additional sub-typing relationships amongst entitlement and responsibility types, provision for these sub-typing relationship requires modification of most programming languages to support the type constructors as primitives, and so was excluded from Chapter Seven. This appendix describes the extended type rules.

A version of Hoare axiomatics [Hoare 1969, Hoare and Wirth 1973] is used to described the type rules as logical inferences. This formalism was employed for this purpose in relation to the sub-typing of updateable values [Connor and Morrison 1992].

A.1 Type Rules for Responsibilities and Entitlements

First, the existing type rules from Chapter Seven are restated axiomatically, First:

$$\frac{\forall T}{RESP[T] \le ENT[T]} \qquad (Equation\ 21)$$

That is, for all types *T*, the responsibility type parameterised by T is a sub-type of the entitlement type parameterised by T. This implies that a responsibility value by be substituted for a value of the corresponding entitlement type, which is reasonable since responsibilities support all of the operations supported by entitlements.

Secondly, there is the sub-typing rule relating each purely indirectly accessed entitlement type to its base type, this is as follows:

$$\frac{\forall T \in \{purely \quad indirectly \quad accessed \quad types\}}{ENT[T] \le T} \qquad (Equation\ 22)$$

This is reasonable since a purely indirectly accessed entitlement type supports all operations applicable to its base type

There is also a sub-typing relationship between entitlement types parameterised by related types, as follows:

$$\frac{\forall S,T : S \le T}{ENT[S] \le ENT[T]} \qquad (Equation\ 23)$$

Recall that the only type specific operation one can apply to an entitlement value (as opposed to its fulfilling value) is demand of the result (whether explicit or implicit). The above type rule is reasonable since any value returned by demand on *ENT*[*S*] will be substitutable in any context where any value returned by demand on *ENT*[*T*] would be substitutable. Combination of this type rule with the first gives a derived rule:

$$\frac{\forall S,T : S \leq T}{RESP[S] \leq ENT[T]} \qquad (Equation\ 24)$$

There are no subtyping relationships between distinct responsibility types. Suppose one had a covariant type rule, as follows:

$$\frac{\forall S,T : S \leq T}{RESP[S] \leq RESP[T]}$$

This would mean that a value of type *RESP*[*S*] would be substitutable for a value of type *RESP*[*T*]. Now, it is legal to supply a value of type *T* to a responsibility value of type *RESP*[*T*], but such a value is of an illegally wide type if supplied to a responsibility value of type *RESP*[*S*], which could occur if this substitution of responsibilities were allowed. Consequently, this substitution can not be allowed and the proposed sub-typing rule is invalid. Now, consider the contravariant alternative:

$$\frac{\forall S,T : S \leq T}{RESP[T] \leq RESP[S]}$$

As with the previous proposal, this permits a substitution that can lead to illegal operations in program execution. In this case the problem is that demand may return an illegally widely typed value. Therefore, the proposed type rule is unsound.

These two rejected rules exhaust the possibilities for sub-typing between distinct responsibility type, and it can be therefore be concluded that no such sub-typing relationship exists.

However, responsibilities are involved in sub-typing relationship in addition to that with entitlements. Recall that an entitlement may be considered a demand-only responsibility. It is also possible to consider a *supply-only responsibility*, or *duty*. A duty that may be supplied a value of type *T* shall be denoted *DUTY*[*T*].

The only operation applicable to a duty is supply of a value of the appropriate parameter type. Given this, a responsibility value is substitutable in a context requiring the corresponding duty, with a type rule as follows:

$$\frac{\forall T}{RESP[T] \leq DUTY[T]} \qquad (Equation\ 25)$$

In fact, it is only be application of this type rule that duty values arise in programs, since it does not make sense to create a duty independently of a responsibility. Instead, a duty is a restricted or abstracted view on a responsibility.

Furthermore, these a relationship amongst distinct Duty types having related parameter types, as follows:

$$\frac{\forall S,T : S \leq T}{DUTY[T] \leq DUTY[S]} \qquad (Equation\ 26)$$

This is a contravariant relationship, whereas the relationship between entitlement types was covariant. This rule is reasonable in allowing a *DUTY*[*T*] to be substituted into a context expecting a *DUTY*[*S*], since any value supplied to a *DUTY*[*S*] will be legal if supplied to a *DUTY*[*T*]. This rule may be combined with the first rule for duties, giving a derived rule:

$$\frac{\forall S,T : S \leq T}{RESP[T] \leq DUTY[S]} \qquad (Equation\ 27)$$

A.2 A Single Rule

The discussion in the previous section included the two type constructors from Chapter Seven and added a third. Another approach is to use a single type constructor: for responsibilities, as follows:

META-RESP[*D*,*S*]

In this case, the first type parameter *D* is the type that may be demanded from the responsibility and the second type parameter *S* is the type that may be supplied to fulfil it.

Given this, the three type constructors in the previous section may be supported as "syntactic sugar", as follows:

RESP[*T*] is equivalent to *META-RESP*[*T*,*T*]

ENT[*T*] is equivalent to *META-RESP*[*T*,*NOTHING*]

DUTY[*T*] is equivalent to *META-RESP*[*NOTHING*,*T*]

Here, the type *NOTHING* is a type having zero values. This type is distinct from the (almost) universal sub-type, often denoted NULL, which has single value (also denoted NULL), which is a legal value for all types except *NOTHING*. The type *NOTHING* is a sub-type of the type *NULL*, and is the only such sub-type.

In addition to the support of existing types, the META-RESP constructor can support types META-RESP[*T*,*S*], where *T* and *S* are distinct types and *S* is a sub-type of *T*. Whether or not this increased flexibility is at all useful is unclear. However, META-RESP is conceptually useful; in consolidating the type rules. Consider:

$$\frac{\forall S,S',T,T' : S \leq T, S' \leq T'}{META-RESP[T,S'] \leq META-RESP[S,T']} \qquad (Equation\ 28)$$

This consolidates the four axioms given in Equations 21, 23, 25 and 26 (and by implication the derived rules in Equations 24 and 27) into a single rule that includes sound sub-typing relationships that are not covered by the original set of rules.

Bibliography

[Achauer 1993] **Achauer, B.**, "The DOWL Distributed Object-oriented Language", *Communications of the ACM*, Vol. 36, No. 9, pp. 48–55, September 1993.

[Acetta *et.al.* 1986] **Acetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M.**, "Mach: A New Kernel Foundation for UNIX Development" in *Proceedings of the Summer USENIX Conference*, pp. 93-112, July 1986.

[Agha 1986] **Agha, G. A.**, ACTORS: A Model of Concurrent Computation in Distributed Systems, The MIT Press, 1986.

[Andrews 1991] **Andrews, G. R.**, "Paradigms for Process Interaction in Distributed Programs", *ACM Computing Surveys*, Vol. 23, No. 1, pp. 49-90, March 1991.

[Arnold and Gosling 1997] **Arnold, K. and Gosling, J.**, "The Java Programming Langauge" 2nd Edition, Addison-Wesley, December 1997.

[Ashenden, Detmold and McKeen 1994a] **Ashenden, P. J., Detmold, H. and McKeen, W. S.**, "Execution of VHDL Models Using Parallel Discrete Event Simulation Algorithms", *VLSI Design*, Vol. 2, No. 1, pp. 1-16, 1994.

[Ashenden, Detmold and McKeen 1994b] **Ashenden, P. J., Detmold, H. and McKeen, W. S.**, "A centralized queue parallel simulator for VHDL", in *Proceedings of International Conference on Simulation and Hardware Description Languages*, Tempe, Arizona, pp. 161-166, 1994.

[Atkinson *et.al.* 1983] **Atkinson, M. P., Bailet, P. J., Chisolm, K. J. Cockshott, W. P. and Morrison, R.**, "An App.roach to Persistent Programming", *The Computer Journal*, Vol. 26, No. 4, pp. 360-365, December 1983.

[Birell *et.al.* 1982] **Birrell, A. D., Levin R., Needham, R. M. and Schroeder, M. D.**, "Grapevine: An Exercise in Distributed Computing", *Communications of the ACM*, Vol 25, pp. 260-274, 1982

[Birrell and Nelson 1984] **Birrell, A. D. and Nelson, B. J.**, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 39–59, ACM, February 1984.

[Black *et. al.* 1986] **Black, A., Hutchinson, N., Jul, E. and Levy, H.**, "Object Structure in the Emerald System", in *Proceedings of OOPSLA'86,* pp. 78–86, ACM, Portland, Oregon, September 29-October 2, 1986.

[Bobrow *et.al.* 1986] **Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M. and Zdybel, F.**, "CommonLoops: Merging Lisp and Object-oriented Programming" in *Proceedings of the Conference on Object Oriented Programming Systems, Languages and App.lications, ACM SIGPLAN Notices*, Vol. 21, No. 11, November 1986.

[Bobrow *et.al.* 1988] **Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G. and Moon, D. A.**, *Common Lisp Object System Specification X3J13* in *ACM SIGPLAN Notices*, Vol. 22 (Special Issue), September 1988.

[Bogle and Liskov 1994] **Bogle, P. and Liskov, B.**, "Reducing Cross Domain Call Overhead Using Batched Futures", *Proceedings of OOPSLA'94, ACM SIGPLAN Notices*, Vol. 29, No. 10, pp. 341–354, ACM, Portland, Oregon, October 1994.

[Brown and Kindel 1996] **Brown N. and Kindel, C.**, *Distributed Component Object Model Protocol – DCOM/1.0*, Microsoft Corporation, 1996.

[Budhiraja *et.al.* 1993] **Budhiraja, N., Marzallo, K., Schnieder, F. B. and Toueg, S.**, "The Primary-Backup App.roach", in *Distributed Systems, 2nd Edition*, (ed. S. Mullender), ACM Press Frontier Series, Addison-Wesley, pp. 199-216, 1993.

[Cardelli and Wegner 1985] **Cardelli, L. and Wegner, P.**, "On Understanding Types, Data Abstraction and Polymorphism", *ACM Computing Surveys*, Vol. 17, No. 4, pp. 471-523, December 1985.

[Caromel 1991] **Caromel, D.**, *Programmation parallele asynchrone et imperative: etudes et propositions*, Doctorat de l'Universitè de Nancy I, Spècialitè Informatique, Febuary 1991.

[Caromel 1993] **Caromel, D.**, "Toward a Method of Object–Oriented Concurrent Programming", *Communications of the ACM*, Vol. 36, No. 9, pp. 90–102, September 1993.

[Caromel, Belloncle and Roudier 1996] **Caromel, D., Belloncle, F. and Roudier Y.**, "The C++// System" in *Parallel Programming Using C++*, G. Wilson and P. Lu editors, MIT Press, 1996.

[Caromel and Detmold 1995] **Caromel, D. and Detmold, H.**, "Notes on Discussions Concerning Eiffel //, Wait-by-necessity and Batched Futures", February 27-March 3, 1995.

[Caromel and Vayssière 1998] **Caromel, D. and Vayssiere, J.**, "A Java Framework for Seamless Sequential, Multi-threaded, and Distributed Programming" in *ACM Workshop 'on Java for High-Performance Network Computing*, Stanford University, Palo Alto, California, February 28—March 1, 1998, pp. 141-150.

[Caromel, Klauser and Vayssière 1998] **Caromel, D., Klauser, J. and Vayssiere, J.**, "Towards Seamless Computing and Metacomputing in Java", Concurrency – Practice and Experience, 10(11–13), pp. 1043-1061, Geoffrey C. Fox (ed), Wiley & Sons, Ltd, September-November 1998

[Chang and Lange 1996] **Chang, D. T. and Lange, D. B.**, "Mobile Aglets: A New Paradigm for Distributed Object Computing on the WWW", *ECOOP'96 Workshop: Towards the Intergration of WWW and Distributed Object Technology.*

[Connor and Morrison 1992] **Connor, R. C. H. and Morrison, R.**, "Subtyping Without Tears" in *Proceredings of the 15th Australian Computer Science Conference*, Hobart, Australia, pp. 209-225, 1992.

[Dasgupta *et.al.* 1990] **P. Dasgupta, R.C. Chen, S. Menon, M.P. Pearson, R. Ananthanarayan, U. Ramachandran, M. Ahamad, R.J. LeBlanc, W.F. App.elbe, J. Bernabeau, P. Hutto, Y. Khalidi and C. Wilkenloh**, "The Design and Implementation of the Clouds Distributed Operating System", *Usenix Computing Journal*, Vol 3, No 1, 1990.

[Dearle, Rosenberg and Vaughan 1991] **Dearle A., Rosenberg J. and Vaughan F.**, "A Remote Execution Mechanism For Distributed Homogeneous Stable Stores" in, *Proceedings of the 3rd International Workshop on Database Programming Languages*, Nafplion, Greece, Morgan Kaufmann (San Mateo,California) pp. 125-138, September 1991.

[DEC *et.al.* 1980] **Digital Equipment Corporation, Intel Corporation and Xerox Corporation**, *Ethernet, a local area network: data link layer and physical layer specifications version 1.0*, Digital Equipment Corporation, Intel Corporation and Xerox Corporation, 1980.

[Detlefs 1995] **Detlefs, D. L.**, "An Overview of the Extended Static Checking System", Research Report, Digital Equipment Corporation Systems Research Center, 1995.

[Detmold 1995] **Detmold, H.**, "Efficient Generalised Futures", in *Proceedings of the First IDEA Workshop*, Coogee Beach Holiday Inn, Coogee Beach, Sydney, New South Wales, Australia, February 19-21, 1995, A. L. Brown (ed).

[Detmold 2000] **Detmold, H.**, "Ambassadors in an Orbital Super-Cluster", in *Proceedings of the Seventh IDEA Workshop*, Whalers Inn, Victor Harbour, South Australia, Australia, February 4-7, 2000.

[Detmold and Oudshoorn 1996a] **Detmold, H. and Oudshoorn, M. J.**, "Supporting the Contract Model for Distributed Programming", in *Proceedings of the Second IDEA Workshop*, The University of Melbourne, Parkville, Melbourne, Victoria, Australia, January 29, 1996, A. J. Hurst and A. L. Brown (eds).

[Detmold and Oudshoorn 1996b] **Detmold, H. and Oudshoorn, M. J.**, "Responsibilities: Supp.ort for Contract Based Distributed Programming", in *Proceedings of the 19th Australasian Computer Science Conference*, Melbourne, Australia, January 31-February 2 1996, pp. 224-233.

[Detmold and Oudshoorn 1996c] **Detmold, H. and Oudshoorn, M. J.**, "Communication Constructs for High Performance Distributed Computing", in *Proceedings of the 19th Australasian Computer Science Conference*, Melbourne, Australia, January 31-February 2 1996, pp. 252-261.

[Detmold, Hollfelder and Oudshoorn 1999] **Detmold, H, Hollfelder, M. and Oudshoorn M. J.**, "Ambassadors: Structured Mobility in Worldwide Distributed Systems", in *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, Austin, Texas, May 31-June5, 1999.

[Dijkstra 1968] **Dijkstra, E. W.**, "Goto Statement Considered Harmful", *Communications of the ACM*, Vol. 11, No. 3, pp. 147-149, March 1968.

[Eppinger, Mummert and Spector 1991] **Epp.inger J. L., Mummert L. B. and Spector, A. Z.**, "Camelot and Avalon: a distributed transaction facility", Morgan Kaufmann, 1991.

[Eswaran *et.al.* 1976] **Eswaran, K. P., Gray, J. N., Lorie, R. A. and Traiger, I. L.**, "The Notions of Consistency and Predicate Locks in a Database System", *Communication of the ACM*, Vol 19, No. 11, pp. 624-633, November 1976.

[Ferenczi 1995] **Ferenczi, S.**, "Guarded Mehods vs. Inheritance Anomaly---Inheritance Anomaly Solved by Nested Method Calls", *ACM SIGPLAN Notices*, Vol.30, No.2, pp. 49-58, February 1995.

[Fischer, Lynch and Paterson 1985] **Fischer, M. J., Lynch N. A. and Paterson, M. S.**, "Impossibility of Distibuted Consensus with One Faulty Process", *Journal of the ACM*, Vol 32, No. 2, pp. 374-382, 1985.

[Fowler 1996] **Fowler, R. J.**, "The Complexity of using Forwarding Addresses for Decentralized Object Finding" in *Proceedings of the 5th Symposium on Principles of Distributed Computing*, 1986.

[Frølund and Agha 1993] **Frølund, S. and Agha, G.**, "A Language Framework for Multi-Object Coordination" in *Proceedings of ECOOP'93, Lecture Notes in Computer Science 791*, Kaiserslautern, Germany, pp. 346-360, Springer-Verlag, , July 1993.

[Gentleman 1981] **Gentleman, W. M.**, "Message Passing between Sequential Processes: the Reply Primitive and the Administrator Concept", *Software – Practice and Experience*, Vol. 11, pp. 435-466, 1981.

[Goldberg *et. al.* 1999] **Goldberg, I., Gribble, S. D., Wagner, S. and Brewer, E. A.**, "The Ninja Jukebox", in *Proceedings of the 2nd USENIX Symposuim on Internet Technologies and Systems*, October 1999.

[Gong 1998] **Gong, L.**, *Java Security Architecture (JDK 1.2)*, Sun Microsystems, 1998

[Gosling and McGilton 1995] **Gosling, J. and McGilton, H.**, "The Java Language Environment: A White Paper", *Technical Report*, Sun Microsystems, 1995.

[Gray and Reuter 1992] **Gray, J. N. and Reuter, A.**, "Transaction Processing: Techniques and Concepts", Morgan Kaufmann, 1992.

[Grimshaw 1993] **Grimshaw, A. S.**, "Easy to Use Object-Oriented Parallel Programming with Mentat", *IEEE Computer*, pp. 39-51, May 1993.

[Grimshaw *et.al.* 1997] **Grimshaw, A. C., Wulf, W. A. and the Legion team**, "The Legion Vision of a Worldwide Virtual Computer", *Communications of the ACM*, Vol. 40, No. 1, January 1997.

[Guttag 1977] **Guttag, J. V.**, "Abstract Data Types and the Development of Data Strctures", *Communications of the ACM*, Vol. 20, No. 6, pp 396-404, June 1977.

[Hadzilacos and Toueg 1993] **Hadzilacos, V. and Toueg, S.**, "Fault-Tolerant Broadcasts and Related Problems", in *Distributed Systems, 2nd Edition*, (ed. S. Mullender), ACM Press Frontier Series, Addison-Wesley, pp. 97-145, 1993.

[Halstead 1985] **Halstead, R.**, "Multilisp: A Language for Concurrent Symbolic Manipulation", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 4, pp. 501-538, October 1985.

[Hamilton and Kougiouris 1993] **Hamilton, G. and Kougiouris, P.**, "The Spring Nucleus: A Microkernel for Objects" in *Proceedings of the 1993 Summer USENIX Conference*, pp. 69-79, December 1993.

[Hoare 1969] **Hoare, C.A.R.**, "An Axiomatic Basis for Computer Prgramming", *Communications of the ACM*, Vol 12. No. 10, pp. 576-580, 583, October 1969.

[Hoare 1978] **Hoare, C.A.R.**, "Communicating Sequential Processes", *Communications of the ACM*, Vol 21, No. 8, pp. 666-677, August 1978.

[Hoare 1985] **Hoare, C.A.R.**, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[Hoare and Wirth 1973] **Hoare, C.A.R. and Wirth, N.**, "An Axiomatic Definition of the Programming Language Pascal" *Acta Informatica*, Vol. 2, pp. 335-355, 1973.

[Hollfelder 1998] **Hollfelder, M.**, "Implementation of the Ambassadors System to Minimize Latency on World-wide Distributed Systems", Honours Project Report, Department of Computer Science, The University of Adelaide.

[Hollfelder, Detmold and Oudshoorn 1999] **Hollfelder, M., Detmold, H. and Oudshoorn, M. J.**, "A Structured Communication Mechanism using Mobile Objects as Ambassadors", *Australian Computer Science Communications*, Volume 21, Number 1, Springer, pp. 265-276, February 1999.

[IEEE 1996] **Institute of Electrical and Electronics Engineers, Inc.**, "Information Technology—Portable Operating System Interface (POSIX®)–Part 1: System App.lication: Program Interface (API) [C Language]", IEEE/ANSI Std 1003.1, 1996 Edition, 1996.

[ISO 1981] **International Organization for Standardization**, "Open Systems Interconnection Basic Reference Model", ACM SIGCOMM, Vol. 11, No. 2, pp. 15-65, 1981.

[ISO 1992] **International Organization for Standardization**, *Information Technology – Database Languages – SQL International Standard ISO/IEC 9075:1992*, International Organization for Standardization, 1992.

[ISO 1995] **International Organization for Standardization**, *Ada95 Reference Manual, International Standard ANSI/ISO/IEC-8652:1995*, International Organization for Standardization, 1995.

[ISO 1998] **International Organization for Standardization**, *Information Technology --- Programming Languages --- C++ International Standard ISO/IEC 14882-1998*, International Organization for Standardization, 1998.

[Jalloul and Potter 1991] **Jalloul, G. and Potter J.**, "Models for Concurrent Eiffel", in *TOOLS 6* (Technology of Object-Oriented Languages and Systems), eds. Potter, J., Tokoro, M. and Meyer, B., Prentice-Hall, Sydney, Australia, pp. 183-191, 1991.

[Jalloul 1994] **Jalloul, G.**, *Concurrent Object-Oriented Systems: A Disciplined Approach*, PhD Thesis, Universoty of Technology, Sydney, Australia, 1994.

[Jézéquel and Meyer 1997], **Jézéquel J-M. and Meyer B.**, "Design by Contract: The Lessons of Ariane", *IEEE Computer*, Vol. 30, No. 1, January 1997.

[Kale and Krishnan 1993] **Kale, L. V. and Krishnan, S.**, "CHARM++: A Portable Concurrent Object-Oriented ystem based on C++" in *Proceedings of OOPSLA' 93, ACM SIGPLAN Notices*, Vol. 28, No. 10, pp. 91-108, ACM, Washington, DC, USA, October 1993.

[Kerry Falkner, Coddington and Oudshoorn 1999] **Kerry Falkner, K. E., Coddington, P. D. and Oudshoorn , M.J.**, "Implementing Asynchronous Remote Method Invocation in Java", in Proceedings of Parallel and Real-Time Systems (PART'99), Melbourne, November 1999.

[Khalidi and Nelson 1993] **Khalidi, Y. A. and Nelson, M. N.**, "An Implmenentation of UNIX on an Object-oriented Operating System" in *Proceedings of the 1993 Winter USENIX Conference*, pp. 469-479, January 1993.

[Lamport 1978] **Lamport, L.**, "Time, Clocks and the Ordering of Event in a Distributed System", *Communications of the ACM*, Vol 21, No. 7, pp. 558-565, 1978.

[Lampson 1986] **Lampson, B. W.**, "Designing a Global Name Service", in *Proceedings of the Fifth ACM Annual Symposium on Principles of Distributed Computing*, Calgary, Canada, pp. 1-10, 1986.

[Lampson 1993] **Lampson, B. W.**, "Reliable Messages and Connection Establishment", in *Distributed Systems, 2nd Edition*, (ed. S. Mullender), ACM Press Frontier Series, Addison-Wesley, pp. 251-281, 1993.

[Legion 1998] **The Legion Research Group**, *Legion 1.4 Developer Manual*, Department of Computer Sceince, University of Virginia, November 1998.

[Liskov 1981] **Liskov, B.** "The Argus Language and System", *Lecture Notes in Computer Science* 190, Springer-Verlag, New York, 1981.

[Liskov *et.al.* 1988] **Liskov, B., Bloom, T., Gifford, D., Scheifler, R. and Weihl, W.**, "Communication in the Mercury System" in *Proceedings of the 21st Annual Hawaii Conference on System Sciences*, IEEE, pp. 178-187, January 1988.

[Liskov *et.al.* 1995] **Liskov, B., Curtis. D., Day, M., Ghemawat, S.., Gruber, R. Johnson, P. and Myers, A. C.,**, "Theta Reference Manual", Programming Methodology Group Memo, MIT Laboratory for Computer Science, 1995.

[Liskov *et.al.* 1996] Liskov, B., Adya, A., Castro, M. and Zondervan Q., "Type-Safe Heterogeneous Sharing can be Fast" in *Seventh International Workshop on Persistent Object Systems*, May 1996.

[Liskov, Herlihy and Gilbert 1986] **Liskov, B., Herlihy, M. and Gilbert, L.**, "Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing", in *Proceedings of the 13th Symposium on Priciples of Programming Langauges*, pp. 150-159, St. Petersburg Beach, Florida, January 13-15 1986.

[Liskov and Shrira 1988] **Liskov, B. and Shrira, L.**, "Promises: Linguistic Supp.ort for Efficient Asynchronous Procedure Calls in Distributed Systems", in *Proceedings of the Sigplan'88 Conference on Programming Language Design and Implementation*, pp. 260–267, Atlanta, Georgia, June 22–24 1988.

[Liskov and Wing 1993] **Liskov, B. and Wing, J.**, "Specifications and Their Use in Defining Subtypes", *Proceedings of OOPSLA'93, ACM SIGPLAN Notices*, Vol. 28, No. 10, pp. 16-28, ACM, Washington, DC, USA, October 1993.

[Liskov and Wing 1994] **Liskov, B. and Wing, J.**, "A Behavioural Notion of Sub-typing", *ACM Transactions on Programming Lasguages and Systems*, Vol. 16, No. 6, pp. November 1994.

[Liskov and Zilles 1974] **Liskov B. H. and Zilles, S. N.**, "Programming with Abstract Data Types", Computational Structures Group, Memo No. 99, MIT Project MAC, Cambridge, Mass., 1974.

[Lister 1977] **Lister, A.**, "The Problem of Nested Monitor Calls", *ACM Operating Systems Review*, pp. 5-7, July 1977.

[McCall, Richards and Walters 1977] **McCall, J., Richards, P. and Walters, G.**, *Factors in Software Quality* (three volumes), NTIS AD-A049-014, 015, 055, November 1977.

[Marlin 1980] **Marlin, C. D.**, *Coroutines, Lecture Notes in Computer Science 95,* Springer-Verlag, New York, 1980.

[Matsuoka, Taura and Yonezawa 1993] **Matsuoka, S., Taura, K. and Yonezawa, A.,** "Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-oriented Languages" in *Proceedings of OOPSLA'93, ACM SIGPLAN Notices*, Vol. 28, No. 10, pp. 109-126, ACM, Washington, DC, USA, October 1993.

[Meyer 1988] **Meyer, B.**, *Object-oriented Software Construction*, Prentice-Hall International, Englewood Cliffs, New Jersey, 1988.

[Meyer 1992] **Meyer, B.**, *Eiffel, the Language*, Prentice-Hall International, Englewood Cliffs, New Jersey, 1992.

[Meyer 1997] **Meyer, B.**, *Object-oriented Software Construction*, 2nd Edition, Prentice-Hall International, Englewood Cliffs, New Jersey, 1997.

[Millstein and Chambers 1999] **Millsteinf, T. and Chambers, C.**. "Modular Statically Typed Multimethods", *Sixth International Workshop on Foundations of Object-Oriented Languages* (FOOL 6), San Antonio, Texas, January 23, 1999.

[Milner 1989] **Milner, R.**, *Communication and Concurrency*, Prentice Hall International Series in Computer Science,1989.

[Milner 1993] **Milner, R.**, "The polyadic pi calculus: a tutorial." in *Logic and Algebra of Specification* (ed. Bauer, F.L., Brauer, W. & Schwichtenberg, H.). NATO ASI Series F, Vol. 94, pp. 203-246. Springer-Verlag, 1993.

[Mockapetris 1987] **Mockapetris, P.**, "Domain Names - Implementation And Specification", RFC 1034, 1987.

[Morrison *et.al.* 1989] **Morrison, R., Brown A. L., Connor, R. C .H. and Dearle, A.** "The Napier88 Reference Manual", Universoty of St. Andrews, Technical Report PP.RR-77-89, 1989.

[Nierstrasz 1987] **Nierstrasz, O. M.**, "HYBRID — a Language for Programming with Active Objects", in *Objects and Things*, pp. 15-42, Tsichritzis, D. C. (ed), Centre Universitaire d'Informatique, Geneva, April 1987.

[Nelson 1981] **Nelson, B. J.**, *Remote Procedure Call*, PhD. thesis, Carnegie–Mellon University, 1981.

[OMG 1997] **Object Management Group**, *The Common Object Request Broker: Architecture and Specification, Version 2.1*, Object Management Group, 1997

[OMG 1998] **Object Management Group**, *CORBA Services Specification*, Object Management Group, 1998

[Oracle 1997] **Oracle Corporation**, "Oracle8 Server Concepts, Release 8.0", 1997.

[OSF 1992] **Open Software Foundation**, *Introduction to OSF DCE*, Open Software Foundation, Cambridge, USA, 1992.

[Oudshoorn 1992] **Oudshoorn, M. J.**, "Atlantis: A tools for Language Definition and Interpreter Synthesis", PhD. Thesis, The University of Adelaide, August 1992.

[Papathomas 1992] **Papathomas, M.**, *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming*, PhD. Thesis 2522, University of Geneva, Geneva, Switzerland, 1992.

[Papathomas and Konstantas 1990] **Papathomas, M. and Konstantas, D.**, "IntegratinbgConcurrency and Object-Oriented Programming: An Evaluation of Hybrid" in Object Management, ed. D Tsichritzis, pp. 229-244, Centre Universitaire d'Informatique, University of Geneva, 1990.

[Parnas 1972] **Parnas, D. L.**, "On the Criteria to Be Used in Decomposing Systems in to Modules", *Communications of the ACM*, Vol. 15, No. 12, pp. 330-336, December 1972.

[Parnas 1978] **Parnas, D. L.**, "The Non-Problem of Nested Monitor Calls", *ACM Operating Systems Review*, Vol. 12, No. 1, pp. 12-14, 1978.

[Philippsen 1995] **Philippsen, M.**, "Imperative Concurrent Object-Oriented Languages", *Technical Report TR-95-050*, International Computer Science Institute, Berkeley, California, August 1995.

[Rodrigues, Anderson and Culler 1997] **Rodrigues, S. H., Anderson T. E., and Culler D. E.**, "High-Performance Local Area Communication using Fast Sockets", USENIX'97, 1997.

[Rosenberg *et.al.* 1996] **Rosenberg J., Dearle A., Hulse D., Lindstrom A. and Norris S.**, "Operating System Supp.ort for Persistent and Recoverable Computations", *Communications of The ACM*, Vol. 39, No. 9 pp. 62-69, September 1996.

[Satyanarayanan 1993] **Satyanarayanan, M.**, "Distributed File Systems", in *Distributed Systems, 2nd Edition*, (ed. S. Mullender), ACM Press Frontier Series, Addison-Wesley, pp. 353-383, 1993.

[Schmidt and Chen 1995] **Schmidt, H. and Chen, J.**, "Reasoning about Concurrent Objects", in *Asia-Pacific Software Engineering (*ASPEC'95) Brisbane (6/12/95), IEEE Computer society press (R. Duke, K. Kang), California, 86-95, 1995.

[Schmidt, Harrison and Al-Shaer 1995] **Schmidt, D. C., Harrison, T. and Al-Shaer, E.**, "Object-Oriented Components for High-speed Network Programming" in *Proceedings of the USENIX Conference on Object-Oriented Technologies*, Monterey, California, United States, June 1995.

[Schneider 1993].**Schneider, F. B.**, "Replication Managmenet using the State Machine App.roach", in *Distributed Systems, 2nd Edition*, (ed. S. Mullender), ACM Press Frontier Series, Addison-Wesley, pp. 169-197, 1993.

[Shibayama, Ichisugi and Yonezawa 1990] **Shibayama, E., Ichisugi, Y. and Yonezawa, A.**, "The ABCL/1 User's Guide", in *ABCL: An Object-Oriented Concurrent System*, pp. 233-308, The MIT Press, Cambridge, Massachusetts, 1990.

[Smith 1979] **Smith, R. G.**, "A Framework for Distributed Problem Solving", in *The Proceedings of the 6th International Joint Conference on Artificial Intelligence*, pp. 836-841, 1979.

[Stroustrup 1991] **Stroustrup, B.**, *The C++ Programming Language, 2nd Edition*, Addison Wesley, 1991.

[Stroustup 1994] **Stroustrup, B.**, *The Design and Evolution of C++*, Addison Wesley, 1994.

[Sun 1985] **Sun Microsystems Inc.** *Remote Procedure Call Protocol Specification*, Sun Microsystems Inc. 1985.

[Sun 1997a] **Sun Microsystems Inc.** *Java Remote Method Invocation Specification,* 1997.

[Sun 1997b] **Sun Microsystems Inc.** *Java Core Reflection Specification,* 1997

[Sun 1997c] **Sun Microsystems Inc.** *Multithreaded Prgoramming Guide,* 1997

[Transarc] **Transarc Corporation**, *Introduction to DCE and Encina*, Transarc Corporation White Paper. *Available online at*: http://www.transarc.com/Library/whitepapers/IntroDE/IntroDE.html

[Wegner 1987] **Wegner, P.**, "Dimensions of Object-based Language Design" in *Proceedings of the Second Conference on Object-Oriented Programming Systems, Languages and App.lications*, pp. 168-182, October 1987.

[Wegner *et.al.* 1991] **Wegner, P., Milner, R., America, P. Nierstrasz, O. Tokoro, M. and Yonezawa, A.,**, "Panel: What is an Object" in *ECOOP'91 Workshop on Object-based Concurrent Computing, Lecture Notes in Computer Science 612*, pp. 257-264, 1991.

[Walker, Floyd and Neves 1990] **Walker, E. F., Floyd, R. and Neves, P.**, "Asynchronous Remote Operation Execution In Distributed Systems", in *Proceedings of the Tenth International Conference on Distributed Computing Systems*, pp. 253–259, Paris, France, May/June 1990.

[Walsh *et.al.* 1985] **Walsh, D., Lyon, B., Sager, G., Chang, J. M., Goldberg, D., Kleiman, S, Lyon, T., Sandberg, R. and Weiss, P.**, "Overview of the SUN Network File System", in *Proceedings of the Winter USENIX Conference*, Dallas, Texas, 1985.

[Xerox 1981] **Xerox Corporation**, "Courier: The Remote Procedure Call Standard", Xerox System Integration Standard XSIS-038112, Stamford, Connecticut, 1981.

[Yonezawa *et.al.* 1987] **Yonezawa, A., Shibayama, E., Takada, T. and Honda, Y.**, "Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1", in *Object-Oriented Concurrent Programming*, pp. 55-89, Yonezawa, A. and Tokoro, M. (eds), The MIT Press, Cambridge, Massachusetts, 1987.

[Zondervan 1995] **Zondervan, Q. Y.**, "Increasing Cross-domain Call Batching Using Promises and Batched Control Structures", Laboratory for Computer Science Technical Report 658, MIT, 1995.