ORTHOGONAL PERSISTENCE, OBJECT-ORIENTATION AND DISTRIBUTION

by

Kevin C.F. Lew Kew Lin

Thesis submitted for the degree of
Doctor of Philosophy
in the Faculty of Engineering of
The University of Adelaide

30 September 1999
Department of Computer Science
This work contains no material that has been accepted for the award of any other degree or diploma in any university or 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.

Kevin C.F. Lew Kew Lin
26 February, 2000

NAME: KEVIN C.F. LEW KEW LIN COURSE: DOCTOR OF PHILOSOPHY

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, 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 Libraries, being available for photocopying and loan.

DATE: 26-02-00


dated

To Anna Forrest
Contents

CONTENTS ........................................................................................................... 1
ACKNOWLEDGEMENT ....................................................................................... v
ABSTRACT ........................................................................................................... vii

1. INTRODUCTION ............................................................................................ 1
1.1. THE PERSISTENCE ABSTRACTION ........................................................... 2
1.2. OBJECT-ORIENTED PROGRAMMING .................................................... 2
1.3. DISTRIBUTION .......................................................................................... 3
1.4. THESIS STATEMENT ................................................................................ 5
1.4.1. Context .................................................................................................. 5
1.4.2. Problem Statement ............................................................................... 5
1.4.3. Thesis Goals ......................................................................................... 6
1.5. ROAD MAP ................................................................................................ 6

2. PERSISTENCE AND OBJECT-ORIENTED PROGRAMMING .................. 9
2.1. INTRODUCTION ........................................................................................ 9
2.2. REACHABILITY STORES ........................................................................ 10
2.3. ALLOCATION STORES ........................................................................... 11
2.4. IMPLEMENTING PERSISTENCE ................................................................. 11
2.4.1. PS-algo—A Mammal among the Dinosaurs ........................................ 12
2.4.2. Mammalian Evolution: The Napier88 Programming Language System 13
2.4.3. E and the Exodus Storage Manager .................................................... 16
2.4.4. O++ and Ode ....................................................................................... 18
2.4.5. The Texas Library ............................................................................... 23
2.4.6. ObjectStore ........................................................................................ 24
2.4.7. PJama ................................................................................................ 29
2.4.8. Other Persistent OO Languages ......................................................... 31
2.5. SUMMARY ................................................................................................. 33

3. OBJECT-ORIENTED ORTHOGONALLY PERSISTENT LANGUAGES ...... 35
3.1. INTRODUCTION ......................................................................................... 35
3.2. RELEVANT OBJECT-ORIENTED CONCEPTS ...................................... 36
3.2.1. Some Popular OO Languages .......................................................... 37
3.3. A MODEL OF OBJECT-ORIENTED ORTHOGONAL PERSISTENCE ........ 39
3.4. ORTHOGONALLY PERSISTENT C++ ...................................................... 40
3.4.1. Primary Issues Arising in C++ .......................................................... 40
3.4.2. The Program Development Model .................................................. 46
3.4.3. The Object Model ............................................................................. 52
3.4.4. Pointer Format and Semantics ......................................................... 53
3.4.5. Some Implementation Concerns ....................................................... 54
I am indebted to my supervisor Dr Fred Brown for seeing me through the four years of my doctoral candidature. I really must mention that Fred got me to start investigating orthogonal persistence in C++ only to comment two years down the track, that he always knew C++ was a horrible language, go figure...

In a major undertaking such as PhD studies, the research environment where one spends so much time is of prime importance. I am glad that under Prof Chris Barter, the Computer Science Department here fosters an atmosphere conducive to study. Here I’ll mention Tracey and Nadia who run the Department smoothly. Oh, and Kat, here you go: Thanx to Katrina Elizabeth Kerry Falkner (gee wot a mouthful!) for the numerous helpful pointers on the distributed aspects of Java. If someone knows about Corba, she does. Henry Detmold shed a lot of light on the way Sun’s Java Remote Method Invocation, RMI, works and this has allowed me to understand why RMI behaved as it did in the performance tests described in chapter 7.

Studies aside, behind every struggling doctoral candidate, are many people whose interactions contribute towards retaining the candidate’s sanity. In my case, the following people and dog come to mind: Grahame, Yvonne and Wolfie Blanchard whose home has always been a haven—I can’t recall not being keen to visit them or meet them at church, Keileng aka Peggy, for her long-distance friendship and continual prayers, Pauline for being instrumental as God’s voice in the wilderness and the many others whom I won’t mention for lack of space. Oh well, you know who you are.

All this would not have happened if way back in the real world of 1993, my dear parents had not been upset at me for wanting to accept a managerial position with the British American Insurance Co Ltd. To them, I am grateful.

Then there is Anna, without whose companionship I would not have had the emotional strength to complete this work. Thank you for putting up with so much from me.

Last but not least, all thanks be to Almighty God whose perfect will brought me thus far. Unknown to most people, a major purpose of me doing this degree is to facilitate entry into third-world countries where doors are closing to the Gospel and it has become virtually impossible for missionaries to obtain work visas. I hope that, for having studied that much in a hi-tech area, my qualifications will be attractive to developing countries so I may get in where others have met with difficulty.

For I am not ashamed of the gospel of Christ, for it is the power of God to salvation for everyone who believes...

Abstract

This thesis develops a model of orthogonal persistence for object-oriented languages. A type-safe model based on the Napier88 Language Environment is proposed and as example, a possible design for a persistent version of C++ is explored. The model uses a persistent object store to support orthogonal persistence in the language. Persistent stores are a proven mechanism whereby persistence can be supported in a language environment. To accommodate the needs of distribution, this thesis also develops an extended model of distributed stores suitable for wide-area networks. The base model used for experimentation is Munro's confederated approach pioneered in STACOS, the St. Andrews Confederated Object Stores project.

STACOS showed that it is possible for autonomous Napier88 stores to cooperate over wide areas. However, the autonomy requirement restricted remote operations to copying and browsing. In this thesis, an extended model is developed and implemented to show that there need not be any compromise between autonomy and expressive power. The new model allows the full range of operations defined by a type to be applied to a remote instance of the type. To achieve that, the concept of an *indirection* is proposed and its use is demonstrated.

With the ability to support a virtual global space from autonomous stores, it then becomes feasible to develop large applications whose resource requirement exceeds that available in individual stores. However, such applications are typically complex and when they span several stores, maintenance and evolution becomes difficult. A need therefore arises for some composition approach that structures software into a number of self-contained spaces. These spaces would interact according to a well-defined behaviour in order to achieve the functionality of the entire software. Such an architecture is conducive to better component management and upgrade than would be possible with monolithic software.

Inherent to Munro's confederated approach is the visibility of store boundaries, with each store representing a locality accessible from others. To afford the structuring of software into a number of well-behaved components, a further extension of the model is the concept of a first-class *locality*. A locality is a self-contained object space, impermeable to direct references and useful for organising re-useable components or for general software architecture. With the ability to compose software using self-contained worlds, hyper-programming, a programming style hitherto restricted to single stores, may now span multiple stores.
Chapter 1
Introduction

Ever since the inception of the persistence abstraction [Atkinson et al 83a] in the early 1980s, research
groups worldwide have endeavoured to investigate the support of persistence in their favourite
programming language. Indeed following the creation of PS-algol [Atkinson et al 82] and Napier88
[Dearle et al 89], the past decade or so has seen the emergence of a plethora of persistent languages, with
flavours ranging from the Algols [Wirth & Hoare 66] to the BCPL families [Richards & Whitby-
Stevens 79]. Some of these were purposely created as persistent languages whilst others were persistent
versions of existing languages. In any case, efforts culminating in products such as Multi-Computer Texas
[Blackburn & Stanton 96], PJama [Atkinson et al 96] and ObjectStore [ODI 98], from major academic
and commercial organisations are an indication that persistence is assuming increasing importance in the
software landscape.

The persistence abstraction relieves programmers of the task of writing code to explicitly pack
and unpack data structures, in order to preserve their state in between program runs. Slowly but surely, a
hitherto little used breed of programming languages has started to attract interest and is likely to be in
fashion by the end of this millennium. They are the object-oriented, OO, languages [ACM 92], so called
because they favour programming using objects and families of objects. For many application domains,
object-oriented analysis and design coupled with the use of an object-oriented language may increase
programmer productivity because the language provides constructs that allow the programmer to naturally
and easily model real-life entities identified at the design stage. This is particularly so when the
application under development is large and complex. It can be decomposed into a number of components,
each of which consists of one or more objects implementing the functionality of the component.

Whilst persistence and object-oriented programming help ease the complex development of
large-scale software, the execution of such software is also notoriously demanding on computing
resources. Often, the sheer size and execution requirements call for processing and storage capacity of the
type that is normally available on distributed platforms. On such platforms, several, possibly autonomous
computing elements provide support for the many resource requirements of the execution. In harnessing
these distributed resources, the execution may reach a level of performance that exceeds what would
normally be obtained from a single node.

One other rationale that has driven research into distribution and persistence has been the
physical limitations of single persistent object stores—a persistent object store [Brown 88] being a proven
mechanism to support persistence in a language. One conceptual property of such stores is that their
persistent space is unbounded. However, in implementation terms persistent stores [Brown &
Morrison 92] are ultimately reliant upon some physical device together with low-level software, e.g. disk
and filesystem, that have an upper bound on capacity. To that effect, distribution seeks to overcome this physical limitation by building the persistent object space using a number of nodes each of which has some storage capacity. One obvious approach is to view the persistent object space as being a single computational space spread over a number of cooperating nodes. The persistent store is then the integration of all the nodes operating in unison. Another approach is to take a number of individual stores and provide some user-friendly communication mechanism to glue them into a conglomeration. Later chapters will discuss how these two views lead to two radically different approaches to implementing distributed persistence.

1.1. The Persistence Abstraction

Informally, persistence is an abstraction that allows program items such as primitive values, structured objects and routines, to have arbitrary longevity regardless of the lifespan of the program that created them. Traditionally, the longevity of program items has been bounded by the execution of the program that manipulates them and when execution terminates those items are lost. Most languages do provide facilities to transfer the content of data structures to some permanent storage e.g. an operating system file or a database table. When the data is needed again in the future, it has to be read back in and the structures, and their relationships, re-created. This packing and unpacking of data is a programmer activity and the amount of code to achieve this has been found to be as much as 30% of the application code [IBM 78].

Unlike traditional programming systems, a persistent language provides mechanisms for the programmer to specify which structures are to outlive program execution. Programmers need not worry about how these items are retained, they just are. The persistent language also affords the programmer the ability to refer to existing persistent data in much the same way as any other data structure [Atkinson et al 83a]. In short, the persistence abstraction moves all concerns about data storage and organisation away from the programmer and makes them a responsibility of the run-time environment. Since in any non-persistent language, the responsibility of managing data ultimately rests on the programmer, the early persistence research pioneers have posited that it should be possible to transfer that responsibility to the run-time environment, thereby converting the language into a persistent one.

When the above transfer happens, the run-time environment is called upon to provide a higher level of support than is the case with most existing programming systems. In so doing, it is hoped that software production costs would be reduced, not only through the use of a persistent language, but by relying on integrated persistent software environments [Atkinson & Morrison 86] of the kind envisioned in the FIDE project [FIDE 90]. The diagram in Figure 1-1, reproduced with slight modifications from [Connor 90] illustrates the targeted model of development environment.

1.2. Object-Oriented Programming

1982 saw the formulation of a prediction that,

*object-oriented programming will be in the 1980s what structured programming was in the 1970s.*

Rentsch, 1982.

Whilst object-oriented programming, solely on the strength of the object paradigm, has not really made as large an impact in the 1980s as structured programming did in the 1970s, there is indication that
awareness, if not definite interest, in object-oriented languages is on the increase. This phenomenon can be traced to the fact that in the Java programming language [Arnold & Gosling 96], the object paradigm has been made available together with other attractive features such as architecture neutrality, distributed programming, security and familiarity of syntax for C/C++ programmers. Indeed over the past few years the object-oriented paradigm has evolved to become the software architecture of choice for a broad range of applications.

Object-orientation is a term used in different although related contexts. There are three main phases of the software life cycle where the term is commonly used. They are the so-called analysis and design phases, hence the terms object-oriented analysis, OOA, and object-oriented design, OOD, and the actual program development phase, i.e. object-oriented programming, OOP. In any one instance of the software life cycle, OOA and OOD need not be followed by OOP although a natural evolution of the development process is to carry through the implementation of the design using an OOP language.

In this thesis object-orientation is used in the third context—that of programming with objects and their behaviour, using a language that provides constructs to do so. A more succinct framework is developed in later chapters.

1.3. Distribution

Distribution may be described as the spreading of computing activities over separate processing units that are linked by some communication medium. In our context, the distribution visible at the software level need not directly match the underlying hardware arrangement. In other words, for the purpose of this thesis, it matters not whether the resources supporting distribution in the software, e.g. the occurrence of
multiple threads of activity, is actually hosted by a single machine. This will become more obvious as later chapters introduce the model of distribution that is of relevance to this thesis.

In most cases however, a distributed system is one where physically separate processors cooperate towards providing resources that would otherwise not be easily available, either because the resources are needed by a community of potentially concurrent users or because the nature of the problem-solving favours a distributed algorithm.

Distributed systems are implemented on a variety of platforms and each platform is characterised by various features which determine its place in the so-called taxonomy of concurrent architectures [Flynn 66]. Three main features whereby distributed systems are characterised are:

- Memory model. Is the memory available shared by all processors or does each processor own local memory?
- Processor Coupling. How closely do the processors cooperate in the computation? It is generally observed that coupling tends to be related to the physical proximity of the processors.
- Communication. Here, latency, bandwidth and reliability are the major differentiating factors.

Currently, in the distributed hardware arena, the architecture that appears to be surviving most easily and is increasing in versatility, is the modest LAN-based workstation cluster [Anderson et al 95]. It is only a matter of time, as wide-area networking technology become more affordable, that the typical distributed architecture will feature a number of computers connected together using a combination of local and wide area arrangements [Tanenbaum 88].

Distributed systems are not new and as mentioned above, they trace their origins from hardware solutions. However, currently the most popular form of distribution is implemented in software and is largely hosted on hardware that was not primarily built for intensive distributed interaction. Here, we have the World-Wide Web [CERN 90] in mind. The increasing popularity of the Web appears to be very much a result of the distributed landscape described above—a multitude of computers loosely connected by a mixture of LAN and WAN technology. Some properties that the Web displays are:

- Even the most modest hardware can participate on the Web. Hardware capacity only needs to match the aspiration of the human operator. For a person who only wants to access and use resources, a personal computer will often suffice whereas a person who wants to offer Web resources to the world needs server-class hardware.
- Service providers offer resources that they feel they can afford to support for whatever length of time they so wish. In so doing, providers grant users the privilege to access their resources, e.g. the contents of a Web page. When the provider deems that the service should be dropped, the privilege is no longer available. In other words, no user may claim a right to access a provider's resources.
- It therefore follows that each service user is responsible for seeking alternative resources should the service provider that they target be unable to support their request. Such a distributed responsibility ensures that for as long as they need it, people make a constructive effort to keep the Web operational.

In view of the above, it would appear that in the future of distributed computing, loosely coupled processing elements, each with local memory and linked by high-latency low bandwidth connectivity, will be prevalent. Therefore, some of the work done as part of this thesis involves the formulation of a distribution model suitable for the type of distributed environment described above.
1.4. Thesis Statement

Having overviewed the three areas of persistence, object-oriented programming and distribution above, this thesis statement now presents the context in which the work described in the rest of the thesis was performed. The problem at hand is also identified and the goals that this work aims to achieve are briefly stated. These goals are re-examined in chapter 9.

1.4.1. Context

The context relevant to the work described here centres on the following three observations:

- Persistence is a useful abstraction that is proven to improve programmer efficiency by taking responsibility for the storage of data.
- Object-orientation is assuming increasing popularity in the programming language community.
- The influence of distributed environments such as the Internet is become stronger and more widespread not only among IT professionals but also with the general public.

In view of the above, the work described in this thesis initially focuses on the provision of orthogonal persistence for object-oriented languages. In particular, it investigates the feasibility of adapting the Napier88 programming language model to a language that supports object-oriented programming, namely C++.

With prolonged use, a persistent store tends to increase its object population as more and more program runs leave behind the results of their computation. Whilst a conceptual property of the persistent object store is its unbounded capacity [Brown 88], the reality is that the underlying storage medium has technological limitations. Therefore, current research has been investigating the spreading of persistent spaces over distributed environments. In the light of this, an additional context to this thesis is that of distribution and the study of models of distributed persistence. This context is especially relevant considering the rate at which the Internet is spreading. It will not be long before isolated, i.e. non-networked, computers will be the exception rather than the rule.

A third aspect of this thesis deals with the provision of a mechanism to support the logical partitioning of a persistent space into self-contained units. As persistent stores become increasingly used in large scale systems, the management of such volumes of persistent data becomes an important issue. This issue may be exacerbated in a distributed environment. To that effect, this thesis posits that together with distribution, it is desirable to provide mechanisms for structuring a large and potentially distributed persistent space.

1.4.2. Problem Statement

Orthogonal persistence support for procedural languages is firmly established in view of the success of PS-algol and Napier88 in convincing the research community that orthogonal persistence is a desirable property of data and of behaviour. With respect to object-oriented languages, this is less clear. The persistence research literature seems to indicate that researchers are still actively developing new models or at least adapting existing models to be used for such languages.

Distributed persistence is another current research area where there is much activity. Research into distributed persistence seeks to alleviate the storage bottleneck that is usually present in centralised persistent object stores.
Given the above state of affairs, there is a need to formulate a model of orthogonal persistence suitable for object-oriented programming in a distributed context. In addition, related issues need to be taken into account. Some of them are:

1. The need to allow the distributed space to scale up for as long as there are additional persistent stores willing to join in the distribution.
2. A structuring mechanism when a persistent space is distributed over a large network. There is a need for some containment mechanism to allow the space to be partitioned into autonomous regions. These regions can thereafter be independently managed under different policies, especially if they are owned by different organisations.
3. How a computation in one autonomous region will pass information to a cooperating region without jeopardising autonomy and safety.

1.4.3. Thesis Goals
This thesis aims to show that orthogonal persistence can be supported in an object-oriented language such as C++. To achieve that, the intent is to start with a model such as the Napier88 programming language environment and adapt it to C++. Ideally this model allows objects, whether data, code or class information, together with executable programs to persist. Class information will need to include both data layout and method code. The model will be reliant upon a generic persistent store architecture [Brown & Morrison 92], to provide a single persistent space.

Single persistent stores are known to be limited in size by the underlying storage system, e.g. the filesystem, on which they rely. To that effect, the issue of distribution as a means to support infinite storage capacity is also addressed. One other goal is to develop a model of distributed persistence able to support the cooperation of physically distant persistent object stores. In this model, a persistent store is able to invoke services offered by the other stores in the distribution. With remote invocation, structured information is often passed between interacting stores and the suitability of current parameter passing conventions used by existing distributed systems is examined. The thesis also investigates the possibility of parameter passing with true by-reference semantics but where direct references will not be allowed to span the distributed space.

Distributed persistent spaces have the potential to span very wide geographical areas. In this kind of environment, management of the persistent stores that compose this environment can become complex. This is especially so when several organisations agree to cooperate without having to relinquish control of their site and ownership of their resources. In view of that, this thesis also aims to develop a mechanism to support the existence of regions within the distributed space. This mechanism will need to allow the regions to exhibit attractive properties such as autonomy, well-defined boundaries, stability and fault-tolerance.

Finally this thesis also aims to show that it is feasible to implement all the concepts or models proposed.

1.5. Road Map
Chapter 2 introduces orthogonal persistence and persistent object stores. It contains mostly background material and is aimed at providing a starting context for the thesis. Chapters 3 and 4 further examine orthogonal persistence in the light of two areas of interest to this thesis, that of object-oriented languages, and that of distribution. Chapter 3 advocates a model of orthogonal persistence suited to object-oriented
languages and proposes possible designs for some such languages. In particular, C++ [Stroustrup 93] is examined and an early hypothesis by [Atkinson & Morrison 83] is tested: that it should be possible to add persistence to a language with minimal disturbance to the language definition. Two reasons underlie the choice of C++ as the vehicle for experimentation. They are (i) the fact that because it has its origin in C [Kernighan & Ritchie 78], C++ is the first widely popular language that offers object-oriented features and, (ii) easy availability of compilers, both of which have led to a fairly large community of C++ literate people in Computer Science.

Distribution constitutes the bulk of the work described here and from chapter 4 onwards, the thesis refines the distributed context. A model of confederated persistent object stores is developed in chapter 5. Chapters 6, 7 and 8 describe an implementation of the model, preliminary performance figures and future possibilities before chapter 9 concludes on the contribution that this thesis has made and re-examines the thesis goals.

On to the thesis now...
Chapter 2
Persistence and Object-Oriented Programming

2.1. Introduction

The persistence of a programming object is defined to be the object’s longevity in a computing environment, where it is accessible for as long as needed by computations [Atkinson et al. 1983a]. Here the term programming object is taken to mean any program item that can be referenced using commonly available constructs in the language used to write the program. A persistent language is a language that provides support to abstract over the differences in longevity of programming objects. It therefore follows that in such a language programmers are relieved from having to deal with the packing and unpacking of data structures.

The most general form of persistence whereby any object, regardless of type, may persist is known as orthogonal persistence [Atkinson et al. 1983b, Atkinson & Morrison 1995]. In an orthogonally persistent system any object, including type information and method code, may persist. The benefits of orthogonal persistence are already well documented and widely accepted [Atkinson & Morrison 1984, Brown 1988, Dearle 1988, Kirby 1992, Munro 1993, Atkinson et al. 1996]. Suffice to say that orthogonal persistence affords the following advantages,

- increased programmer efficiency through the provision of simpler storage semantics,
- extended type security to cover the life-time of the persistent object,
- automatic translation between an object’s run-time format and its long-term storage format thereby removing the need for explicit translation between formats,
- enforcement of referential integrity of complex object graphs,
- increased code sharing and reuse, thus promoting programmer productivity through less coding,
- the ability to support hyper-programming, a versatile programming activity where the checking of access to persistent objects and their linking into the program under development may largely happen during program composition [Kirby 1992, Farkas et al. 1992].

The ability of a persistent system to support orthogonal persistence has traditionally been associated with that system’s mechanism of identifying persistent objects. There are three main mechanisms of persistent data identification [Dearle 1987, Brown 1988, Richardson & Carey 1989, Hosking 1991, Dar et al. 1993]. They are,

- Reachability-based persistence,
- Allocation persistence and its derivatives, and,
- Core-dumping.
The core-dumping approach to persistent data identification is only mentioned here for the sake of completeness. Extensive treatment of the limitations of core-dumping has already been documented and thus, core-dumping is not considered any further in this thesis. For more detailed work on core-dumping issues, see [Goldberg & Robson 83, Teitelman et al 78 and Brown 88].

As this chapter will go on to show, the mechanism of persistent data identification adopted by a store influences the way persistence is supported in the programming language that uses the store.

2.2. Reachability Stores

Reachability-based persistence identifies an object to be persistent on the basis of its reachability from some predefined “pointer” commonly known as a root of persistence. In a reachability-based store, persistent objects would be organised as directed graphs, with at least one root as entry point to each graph. Under this arrangement, the persistent store population is then defined to be the collective closure of all the roots of persistence in the store. An object obj is therefore retained if at the time of determining the persistent state of the store, obj is found to be transitively reachable from one such root.

The reachability approach has its foundation in graph theory where reachability and its property of transitivity ensure that all vertices within the closure of a digraph have a directed walk from an entry point of the graph. Given a digraph G and a vertex v, a reach algorithm when applied to G, outputs a list of all the vertices reachable from v.

Therefore, reachability stores either provide,

- at least one root of persistence, the identity of which never changes, or
- a mechanism for program executions to dynamically create roots of persistence as entry points to the persistent space of the store. An associated mechanism to access existing roots is usually provided to complement the creation of roots.

With the ability to access predefined roots or to dynamically create new ones, programs executing in the store may then include new objects or access existing objects from the roots. Thereafter, computing the transitive closures determines the persistence of objects in the store. Unreachable objects may later be garbage-collected at a convenient time—we note that automatic garbage-collection has always been an integral feature of reachability stores.

The computing of root closures usually take place whenever a stable state of the persistent store needs to be determined. This activity may occur at both store commit and at garbage-collection time. During the computing of root closures, any existing object whose state has been modified, is updated in store. Additionally, it is likely that the traversal algorithm encounters a new object for which there is no existing store copy. In this case, store space is allocated on-the-fly and the contents of the object are saved in the newly obtained space. To achieve this, the store provides a space allocation routine that gets called by the commit operation or during garbage collection.

It is to be noted that with reachability stores, object deletion and any necessary space deallocation is not an explicit operation but it is rather a “side-effect” resulting from an object being outside root closures. As a result, it is easier to decide what is or is not garbage, making collection simpler in reachability stores.
2.3. Allocation Stores

Allocation persistence identifies an object to be persistent based on the existence of some pre-allocated non-volatile space inside the persistent store. An object is retained if such storage space has been allocated and is associated with the said object. As is the case with reachability stores therefore, an allocation store also provides a space allocation routine for obtaining necessary store space. However, a primary difference lies in the time at which allocation happens. Here, space allocation is usually performed at the time of instantiation of the object and the association between object and storage space typically takes the form of a storage “pointer” obtained from the allocation. The space pointed to, is reserved with the intent of copying the state of the object at an appropriate point in time, e.g. during a commit operation. Thereafter, the object persists until an explicit de-allocation removes the association between the object in memory and its non-volatile storage.

The allocation-based approach described above can be termed *eager* allocation since store space is obtained at instantiation time, even if the object is deleted during the same program run. A variation of the technique, which we call the *lazy* approach, has been to notify the storage manager that an object will persist. Thus, instead of pre-allocating non-volatile space, sufficient information about the would-be-persistent object is maintained so that at a convenient time, that information is used to track down the object and to copy its content to non-volatile storage. Such information usually takes the form of the memory address of the object along with the object’s size, and is passed to the store manager at the time of instantiating the object. The store manager then retains this information for later use. In the lazy approach, one would expect the object instantiation to be faster. In addition, a smart implementation will know that it is unnecessary to create store space for deleted objects, at commit time.

Typically, an allocation-based store would be equipped with routines that allocate and de-allocate non-volatile space. Some stores achieve this by providing functions that read from and write to the persistent store [Carey & DeWitt 86] whilst others might rely on a pre-compiler of the language they support, to synthesise such routines for every user type defined by the programmer at the programming language level [Agrawal & Gehani 89].

Over time as computations leave objects to persist in the store, some such objects ultimately become unwanted and need to be disposed of. To date, garbage collection is still a complicated issue in allocation stores. This is because allocation stores tie up persistent data identification with object definition at the programming level, which is an explicit programmer activity. Having done so, allocation stores also associate the removal of unwanted objects in the store with a “delete” operation which is also a programmer responsibility. It therefore follows that garbage accumulates in the store either as a result of careless programming, or because some objects are forgotten over time.

The majority of research projects on allocation stores have investigated persistence support for the C++ programming language or some variant thereof [Richardson & Carey 89, Agrawal & Gehani 89, Singhal et al 92, ODI 98]. There is a straightforward correspondence between the allocation and de-allocation of non-volatile space in the store, and the new and delete operators of C++. This has made allocation persistence a very attractive model for C++ as well as for other languages where object creation/destruction, and the associated memory management are a programmer responsibility.

2.4. Implementing Persistence

The rest of this chapter reviews several implementations of persistent languages and their underlying store. The focus is on how a store’s philosophy of persistent data identification affects the way persistence
is provided in the language supported by that store. The review gives insight into the properties exhibited by persistent systems as a result of the persistent data identification approach they adopted. It has been primarily restricted to those stores that have been used to support object-oriented languages. PS-algol and Napier88 are also included because of their orthogonal persistence model. Certain systems have been singled out for detailed survey because they present peculiarities that contribute to understanding persistent systems design.

It is widely accepted that reachability-based persistence ultimately leads to orthogonal persistence, although this chapter later studies one particular system that is reachability-based yet fails to achieve orthogonal persistence.

In the field of persistence, research issues are not limited to persistent data identification, there is active investigation in the following areas:

- type systems and type persistence,
- binding mechanisms,
- garbage collection, especially in a distributed setting,
- persistent object store design and implementation,
- hyper-programming—which is a composition approach to persistent software.

In addition, there is ancillary research into various issues regarding optimisation and tuning, e.g. object formats, swizzling, faulting techniques and granularity. The discussion in this chapter focuses mainly on persistent data identification and its influence on the provision of orthogonal persistence at the programming language level.

We now examine two persistent systems that achieve orthogonal persistence and rely on reachability to identify persistent data.

### 2.4.1. PS-algol—A Mammal among the Dinosaurs

Persistent systems have sometimes been called "the mammals among the dinosaurs" by researchers at various persistence workshops and conferences [Atkinson 99]. If there is one language that convincingly triggered the dawn of persistence onto the programming language scene, it would have to be PS-algol. This section describes PS-algol and the account is based on [Atkinson et al 82, Atkinson et al 83a&b, Brown 88 and Dearle 88]. PS-algol extends an existing language, namely S-algol [Morrison 79], to incorporate persistence. PS-algol values of any type may persist, with the persistence of heap-based values being decided by their reachability from a set of persistence roots. Essentially, the idea is to provide the illusion of a large persistent heap of objects organised into a number of named databases. Here an object means any S-algol structure, array or string. A program interacts with the heap by first opening one or more databases and then reading/writing objects to and from them. The model of program interaction with databases in PS-algol is the single-writer/multiple-reader approach.

A database is explicitly opened through a call to function open.database, which takes the name of the database, a password and an opening mode, "read" or "write", as string parameters. If successful, the call returns a pointer to a table of roots and a transaction is implicitly started. Having so obtained a table, a program can thereafter access the database population by traversing more pointers that happen to be reachable from a root. At some point during its execution, the program may call procedure commit to make modifications permanent on a database opened for writing. Not calling commit implies abort when the program terminates and automatically closes all databases. Procedure close.database can be called to explicitly close a database whose name is a string parameter to the call.
Depending on the mode specified for opening a database, a shared lock or an exclusive lock is attempted on the database. If this fails, open.database returns a pointer to an error.record structure.

New databases may be created through a call to create.database which returns a table if successful or an error.record if not. Standard operations on tables are also provided in PS-algol. Two simple examples, adapted from [Brown 88], are given below. The first one stores an integer in a newly created database:

```
structure intContainer ( int anInteger )
let db = create.database ( "myDB", "aPassword" )
if db is error.record do
  { write "Can't create database" ; abort }
s.enter ( "myInteger", db, intContainer ( 3 ) )
let ok = commit ( )
if ok is error.record do { write "Can't commit" ; abort }
```

and the second example retrieves the stored integer, for use by the program:

```
structure intContainer ( int anInteger )
let db = open.database ( "myDB", "aPassword", "read" )
if db is error.record do
  { write "Can't open database" ; abort }
let p = s.lookup ( "myInteger", db )
write p ( anInteger )
```

### 2.4.2. Mammalian Evolution: The Napier88 Programming Language System

Insight gained from the success of PS-algol led to the development of Napier88 as a successor to PS-algol. Unlike PS-algol, Napier88 was designed to be an integrated orthogonally persistent programming system. Therefore, whilst PS-algol programs executed against a persistent store, a Napier88 program executes inside its store environment. To that effect, an instance of the Napier88 system consists of the Napier88 persistent programming language and its persistent environment. The system includes a persistent store that is pre-populated with a compiler and standard library procedures amongst other tools. As mentioned above, the PS-algol store consists of a number of databases, each of which is implemented as a file. Difficulty in supporting cross-database references has led to the adoption of a single store file and environment in Napier88.

Regarding the language itself, Napier88 applies the three principles [Atkinson et al 83, Atkinson & Morrison 95] needed to obtain orthogonal persistence. They are the principles of:

- Persistence Independence—program sources look the same regardless of the longevity of the objects they manipulate.
- Type Orthogonality—object longevity is not related to the type of the object.
With respect to Persistence Identification, Napier88 implements persistence by reachability—a single predefined root of persistence acts as access point into the persistent store environment. Thus a program operating in an instance of the Napier88 store environment accesses the root by calling the predefined procedure PS, to obtain the value of the root for that particular instance. Thereafter, the program may cause objects to persist by extending the closure of the root.

Conceptually, the Napier88 system architecture is organised as a number of layers [Brown et al 90]. If present in an actual implementation, each layer presents a higher level of abstraction to the one directly on top. Traditionally, the diagram used to illustrate the architecture is from [Brown 88] and is replicated as Figure 2-1.

In the above diagram, the highest conceptual layer is the programming language level. It is at this level that a program executes and interacts with its home persistent environment. The simple code fragment below illustrates how a Napier88 program could access persistent data and modify it:

```haskell
type employee is structure ( ID : int ; name, position : string ; salary : real )
let ps = PS ( ) ! Obtain the value of PS for this store environment.
project ps as e1 onto employee : begin ! Use e1 as an employee here.
    e1 ( position ) := "CEO" ; e1 ( salary ) := 10000.00
end
default : { } ! Oops, ps wasn’t an employee, do nothing.
```

The root of persistence has type `any`, which is the infinite union of all types that may be defined in the language. Values of any type may be converted to and from type `any` in a type-secure manner. The above code sample shows a value of type `any` being coerced into a value of type `employee`.

![Figure 2-1: The Napier88 layered architecture.](image)
In practice, the root of persistence will usually provide access to data that is much more complex than an employee structure. More complex data and the intricate relationships among them can be organised using the Napier88 environment [Dearle 89] construct. Environments are collections of name-value-type bindings and are first-class objects. They enable the programmer to organise the persistent space in a controlled and type-secure fashion. In a standard release of the Napier88 system, one of the environments that is accessible from PS is User. The following sample code illustrates how environments can be used to localise data—here the code fragment defines a procedure inside User.

```plaintext
type coordinate is structure ( x, y : real )
let ps = PS ( )
project ps as anEnv onto
env : use anEnv with User : env in ! Access the User env here.
begin
  in User let translate = proc ( c : coordinate ; x, y : real -> coordinate )
    coordinate ( c(x) + x, c(y) + y )
  end
default : { } ! Projection failed, do nothing.
```

Having done so, procedure translate is now part of the persistent store environment. It can later be accessed and called as shown below:

```plaintext
type coordinate is structure ( x, y : real )
let ps = PS ( )
project ps as anEnv onto
env : use anEnv with User : env in ! Access the User env here.
  use User with translate = proc ( c : coordinate ; x, y : real -> coordinate ) in
  begin
    let c1 = coordinate ( 1.0, 2.0 )
    let c2 = translate ( c1, 3.0, 4.0 )
  end
default : { } ! Projection failed, do nothing.
```

Whilst the Napier88 approach does away with PS-algol's multiple databases, it does recognise that there often is a need for a program in one store to access other persistent stores. To that effect the Napier88 architecture incorporates a model of distributed persistence to support that need. This aspect of the architecture is examined in greater detail in chapter 4.

So far, the code samples to illustrate the use of orthogonal persistence in Napier88 have been strictly through the use of programming language constructs such as `project .. onto` and `use .. with`. In terms of expressive power and succinctness, this is no different from other constructs available in ordinary languages, e.g. for loops and if statements. A more sophisticated composition technique, known as hyper-programming, is also available for creating Napier88 programs [Kirby 92]. Treatment of hyper-programming is postponed till chapter 8 of this thesis. Suffice to say at this stage that hyper-programming
affords the programmer increased succinctness through less coding, and more safety through earlier program checking.

It is through the two imperative languages PS-algol and Napier88 that persistence first appeared. However, the object-oriented language community was quick to borrow the concept and not long after, a myriad of persistent OO languages started to emerge from research laboratories world-wide. Almost all of them are persistent versions of some existing language such as Smalltalk [Goldberg & Robson 83], Modula-3 [Cardelli et al 89] or C++ [Stroustrup 93], with the majority being persistent versions of C++. Lately, with the prominence of Java [Arnold & Gosling 96] on the programming language scene, the same phenomenon appears to be happening: many persistent Java endeavours have been announced. The remaining part of this chapter examines some of these systems.

2.4.3. E and the Exodus Storage Manager

Exodus [Carey & DeWitt 86] was a multi-faceted project, of which the Exodus Storage Manager, ESM, is the component responsible for supporting persistence. E [Richardson & Carey 89] is the programming language in which programs can be written to run against the Storage Manager. ESM allows storage objects to be allocated in the storage space. Storage objects are an uninterpreted sequence of bytes which may conceptually be of any size. Each of these objects is identified by an object identifier, OID, which is its physical address.

ESM provides an interface via which routines may be called to operate on storage objects. Operation read, given an OID, an offset and a length, causes the Storage Manager to bring the byte sequence identified by the OID into its buffers. The operation returns a user descriptor to the caller. This descriptor contains a pointer to the data in the buffer and de-referencing that pointer accesses the data. A corresponding release operation is available when the data is no longer needed. This operation returns the descriptor to the Storage Manager together with an indication whether the data has been modified or not.

The read-release pair of operations constitute what is known as “pinning” in Exodus terminology. Essentially, pinning is an agreement whereby,

1. the Storage Manager brings in the requested sequence of bytes into its buffers and retains them there for as long as needed by the caller, and
2. the read caller does not attempt to access beyond the pinned region and remembers to unpin the data via a call to release in due time.

Exodus storage objects are organised into files. These are collections of storage objects ordered by OID. Files may be viewed as being disjoint sets with each object residing in one and only one file. Operations on files include creating and destroying an object in a file, and scanning a file to return OIDs of contained objects.

E [Richardson & Carey 1989] is an extension of C++, providing a full range of “database” types that are the persistent complement of the transient C++ data types. They are dbshort, dbint, dblong, dbfloat, dbdouble, dbchar, dbvoid, dbclass, dbsubset, dbunion, pointers to db type objects and arrays of db type objects. db types are used to describe the types of objects in the Storage Manager. E also provides the persistent storage class that is used in conjunction with a db type for making an instance of a db type persistent.

An object may only be persistent if it is an instance of a db type although an object of a db type need not persist. Stack objects persist only if the storage class persistent is specified in the definition expression that instantiates it. A simple example is given below:
dbclass employee
{
    //Whatever data members here...

    public:
    employee(int id, char * name, char * job_description, double salary);
    // etc...
};

int main()
{
    persistent employee an_emp (123, "Jane Citizen", "Programmer", 45.67);
    // Henceforth, an_emp persists as a variable.
    // etc...
}

From the above example, we note that an E object persists as a named variable. In other words, the variable name is a persistent handle on the object in the store. This approach for making objects instantiated on the run-time stack persistent is simple and elegant. To the programmer, the persistent variable is no different from ordinary ones, except that the variable never really goes out of scope from the store’s point of view. Unfortunately a major disadvantage lies in that whilst non-persistent variables on the run-time stack are destroyed as and when they go out of scope, E’s persistent variables never do so and thus are never destroyed. This may lead to garbage accumulation and name space pollution in store.

As mentioned above, E’s view of the persistent store is that of a flat name-value space—there is no persistent heap. So, E provides the collection abstraction to support dynamic instantiation and deletion of objects. A collection is similar to a typed heap where run-time allocation and de-allocation operations may be carried out by the programmer. To achieve that, E overloads the C++ new and delete operators so that it is possible to emulate a persistent heap using a collection defined to store db objects. For a dynamic object to be persistent in E, it must therefore be (i) an instance of a db type and (ii) be allocated within a collection defined to store persistent objects of the db type. For example:

#include <E/collection.h>
dbclass employee { // etc... };
// ... 
persistent collection<employee> a_company;
employee * p1 = new ( a_company ) employee (123, "Jane Citizen", "Programmer", 45.67);

To access existing collections, E provides a scanning class that complements the collection template. An example of how the two are used together is given below.
Deletion of collection-based objects follows the same style as conventional C++, i.e. through the use of the C++ delete operator. If a destructor is defined by the object’s class, the destructor is called as would be expected. Collections can be destroyed as well, in which case all the contained objects are also destroyed. The semantics of object deletion here mirror those of C++ and inherits all its problems, namely dangling references, memory leaks and garbage accumulation.

E was developed with the intention of producing a language in which database systems could be written. Thus the need to specify which objects persist, at instantiation time was not so much of a disadvantage since in database management systems, right from the start there is a clear understanding of which structures will persist and which ones will not.

Several implementations of E compilers of industrial strength have been developed and are still available. However, later research directions in E explored an interpretive approach [Schuh et al 90].

2.4.4. O++ and Ode

O++ [Agrawal & Gehani 1989, Lieuwen 1997] is an upwards compatible extension of C++ and Ode is its underlying object-oriented database system. Whilst E viewed the persistent store as a flat name-value space and did not have an implicit persistent heap, the Ode store functions as a persistent heap to O++ programs. The Ode store groups its persistent objects into clusters each of which represents a type extent. Each cluster contains objects of the same type although there are some “untyped clusters” that contain objects of multiple types as well. In store, a persistent object may reference other persistent ones. A persistent object may exist in two forms in its cluster, depending on whether the object has been referenced or not. All persistent objects have a copy existing on disk. When an object is needed by some program executing against the Ode store, the object is activated and a second copy is created in memory. Each cluster manages its persistent objects and their memory forms. The inverse passivation process synchronises the disk and the memory forms.

To create persistent objects, the Ode object manager proceeds as follows: an object is first created in memory. To make it persistent, it is inserted into a cluster. This insert operation essentially saves the memory address of the object in the cluster. Eventually a write instruction, e.g. when the store commits, creates a disk version of the object and the memory copy is deleted. The Ode store itself does not provide object read and write operations on clusters. It is the O++ compiler that generates code for these operations and includes them in the class definitions used for persistent objects.

Thus for every persistent class defined in an O++ program, the O++ compiler modifies it to add the necessary code needed to interact with the Ode store. These modifications involve:

1. Create an in-memory type descriptor which is used for various purposes including activating objects read from disk. Parts of the descriptor are made persistent when the first persistent object of that type is created.
2. Change all definitions of pointers to persistent objects into objects of class _pref which is a persistent pointer format predefined in the Ode library.

3. Instantiate a cluster for the class.

Object deletion from the Ode store is an operation available as the destructor function of the class PersBase. The destructor is inherited by the persistent class defined by the user. A delete on a class object is thus translated into a call to the destructor to perform clean-up.

In the O++ memory model, memory is explicitly visualised as being of two kinds: volatile and persistent. Persistent memory is made up of a number of databases in much the same way as PS-algol databases. Collectively, the databases define the Ode storage space. Persistent objects are allocated in the Ode store and these continue to exist thereafter whilst volatile objects are only instantiated on the transient heap. Because the language makes an explicit difference between the two kinds of memory, it has to provide a complete range of constructs for specifically handling persistent objects. Operators pnew, pdelete and the special variable pthis are the persistent complement of the normal C++ memory management operators and of the implicit this pointer respectively. Keyword persistent is provided for handling persistent objects and it may be used in two different contexts:

1. When used in the declaration of a class or in a forward declaration of a class, it is a storage class specifier. When the O++ compiler encounters such a declaration, it performs the modifications explained earlier to equip the class so that its objects may persist in the Ode store.

2. When applied to a pointer definition, persistent is a type qualifier and is used in a pointer definition to instantiate a pointer to a persistent object.

Therefore, a typical O++ program would define some persistent class and then, using persistent pointers, instantiate objects using pnew. When the program is compiled, the O++ compiler translates the call to pnew into the insert operation on the class’s Ode cluster. The code fragment below shows how to define a persistent class and create persistent instances of the class.

```c++
#include <ode.h>
persistent class employee ;  // Tell the O++ compiler that instances
class employee  // of this class may persist.
{
    int id ;  // Some private data members, just
    char name [ 50 ], descr [ 50 ] ;  // like ordinary C++.
    double sal ;

    public :
        employee ( int id, char * name, char* job_description, double salary ) ;
        // etc...

} ;
```
int main ()
{
database * db ;
if(( db = database :: open ( "my_database.db" )) == NULL )
{
    cout << "Cannot open my_database.db" << endl ;
    exit ( -1 ) ;
}
employee * ep1 = new employee ( 123, "John Doe", "Janitor", 4.56 ) ;
// A transient employee object has been created and is referenced by ep1.
persistent employee * ep2 = pnew employee ( 123, "Jane Citizen", "Boss", 45.67 ) ;
// Now a persistent employee object has been created and is referenced by ep2.
strcpy ( ep1->descr, "Cleaner" ) ;
ep2->sal = 2345.67 ;
// etc...
delete ep1 ;
// Delete ep1 with the normal delete,
pdelete ep2 ;
// but use pdelete for ep2.
db->close ( ) ;
}

Operator pnew comes in many forms. In its simplest form, it allocates a persistent object in the default database as shown in the code sample above. O++ programs are able to access different databases in a single run. The first database opened by a program is the default database. With multiple databases opened, a program may change the value of the default database or may opt to specify the database using a different form of pnew as shown next.

database * db2 = database :: open ( "database2.db" ) ;
persistent employee * ep = pnew employee ( 123, "Jane Citizen", "Typist", 4.56 ) in db2 ;

To access existing persistent objects inside the Ode store, O++ provides an associative for .. in loop to iterate over all objects of a specified cluster. For example:

persistent employee * ep ;
for ( ep in employee )
{
    cout << ep->id << endl << ep->name << endl ;
    // persistent object and print
    cout << ep->descr << endl << ep->sal << endl ;
    // out the state of the object.
}
// Here the value of ep will be NULL.
Type extents such as employee in the example above contain objects of class employee only. O++ also provides containers that are not restricted to objects of a single class. They are cluster objects and can be used as shown below:

```cpp
database * ua;
if (( ua = database::open("UniAdelaide") ) == NULL) { /* Spit the dummy etc... */ }

trans // Start a transaction block.
{
    cluster Lecturers ("Lecturers", ua);
    persistent employee * p = pnew employee ( /* etc... */ ) in Lecturers;
}
```

Although unlike extents, clusters may contain objects of different types, a for .. in loop that uses say, a persistent employee pointer to iterate over a cluster will only access the persistent employee objects of that cluster. To access all the objects contained in a cluster, a persistent void pointer must be used as loop variable.

Note the trans { ... } construct provided by O++ in the above code fragment. This is the mechanism whereby O++ supports atomic transactions. Exiting a transaction block, whether implicitly by dropping out of the block’s scope or explicitly by executing a break, continue or return, causes Ode to attempt a commit. Apart from database opening and closing, all code that interacts with the Ode store must occur within a transaction block. For simplicity, the trans { ... } was omitted from earlier sample code. Read-only and hypothetical transactions are also supported through the readonly trans { ... } and the hypothetical trans { ... } constructs.

O++ also provides more elaborate forms of the for .. in loop to allow iteration over multiple extents/clusters, to restrict the results of an iteration to those objects that satisfy a boolean expression or to take derived objects into account. For example:

```cpp
// Example of an iteration over two extents/clusters.
for ( ep in employee ; dp in department )
{
    // Produce the cross product of employee and department.
}

// Example of a boolean-controlled iteration.
for ( ep in employee ) suchthat ( ep->sal > 100 )
{
    cout << ep->name << endl;
    // Only search for employees with salaries over 100.
}
```
Dynamic arrays of persistent objects can be allocated using operator *new [ ] or a variant thereof and are treated in much the same way as dynamic arrays in C++. In particular, a pointer to a persistent array refers to the zeroth element of the array. It therefore follows that when a *for .. in* iteration retrieves such an array, the loop pointer variable will reference the zeroth element. There doesn’t appear to be any mechanism to help the O++ programmer determine whether that object is part of an array or not.

Finally another notable feature of O++ is the support available for object versioning. Versioning in O++ is a property of objects, not of classes. Given a persistent object referenced by a pointer ptr, a call to *newvers ( ptr )* returns a pointer to a new version of the object referenced by ptr. Ode will record the relationship between object versions. Functions vroot, vdprev and vlatest are available to navigate version graphs. These functions navigate a version graph by following the implicit links that connect a new version to the immediate previous version. To navigate time-based versions of an object, function vtprev is available. For example, consider the version graph shown in Figure 2-2.

![Figure 2-2: A version graph in the Ode store.](image)

A call to *vdprev* with a pointer to *objj* as argument returns a pointer to *objj* whereas a call to *vtprev* returns a pointer to *objj*. A call to *vroot* on a pointer to any object in the above graph returns a pointer to *objj*. Applying vlatest to *objj* returns *objj* whereas applying vlatest to *objj* or *objj* both evaluate to *objj*.

In addition to the features surveyed so far, O++ also provides facilities to name persistent objects by associating some string identifier with them—this is similar to entering a name-value pair in a PS-algo database. Over the years and with continual development, O++ has deviated from its initial design goal that “language changes should be kept to a minimum”. In particular, the extensions added, e.g. *indexable* and *trans* among other constructs, gear the language more towards database programming and bulk transaction processing compared to pure C++. 

// Example of an iteration over a base extent and its derivations.
persistent class supervisor : public employee /* Derivation from employee base class. */;
persistent class manager : public supervisor /* Derivation from supervisor. */;
persistent class janitor : public employee /* Another derivation from employee. */;
// etc...
for ( ep in all employee ) /* This loop iterates over employee and its derived classes. */

```
persistent class supervisor : public employee { 
  // Derivation from employee base class.
  public employee
  
  persistent class manager : public supervisor { 
    // Derivation from supervisor.
    public supervisor
    
    persistent class janitor : public employee { 
      // Another derivation from employee.
      public employee
      
      // etc...
      
      for ( ep in all employee ) { /* This loop iterates over employee and its derived classes. */
```
2.4.5. The Texas Library

Texas [Singhal et al. 92] is a persistent storage system for C++ and is implemented as a library. By making calls to routines from the library, conventional C++ programs are able to manipulate persistent objects from the Texas store. These routines are specifically provided to create, access and retain persistent objects in the store, as opposed to ordinary objects on the transient heap.

When using Texas, the programmer has the illusion of a large flat address space and interacts with persistent objects as though they were on the heap. The Texas store itself is a reachability store that uses a table of persistent roots to determine the persistent store population. Texas organises its persistent space into pages and the root table is located at a known place in the store—it is the first object in the first page of the store.

Among other functions and macros, the latest release of the library provides the following as an interface for opening and closing a Texas store:

```c
pstore * open_pstore ( store_name ) ;
void close_pstore ( ps ) ;
void close_pstore_without_commit ( ps ) ;
```

and the following to manipulate either named or unnamed persistent objects:

```c
pnew ( ps, type ) ;
pnew_array ( ps, type, count ) ;
is_root ( ps, name ) ;
add_root ( ps, type, name, ptr ) ;
remove_root ( ps, name ) ;
set_root ( ps, type, name, ptr ) ;
get_root ( ps, type, name ) ;
commit_transaction ( ps ) ;
abort_transaction ( ps ) ;
```

In the above interface, `ps` is a pointer to the `pstore` object obtained from a call to `open_pstore`, `type` is the name of the class of the object being created or manipulated, `ptr` is a pointer to the object and `name` is a string used to identify the object.

Given the above interface, the programming model is therefore fairly simple: after opening a persistent store, objects instantiated from a call to `pnew` or `pnew_array` can either be rooted to the store or they can be included in the closure of an existing root. Conversely, existing persistent objects can be accessed directly by name if they are root objects, or indirectly by navigating from one such object.

Texas does not allow the mixture of pointer and non-pointer fields in its persistent unions. This is because type descriptors, from generating debugging information, are obtained from a static analysis of the source yet a union is able to hold data of different types during program execution. Effectively therefore, the union object potentially assumes a different type every time its state changes. Such dynamic change in type identity makes the Texas swizzler unable to correctly identify pointer fields.

Although Texas relies on standard reachability to identify which objects persist, it fails to support orthogonal persistence. Closer analysis reveals that the problem does not lie in its model per se but in the
library approach of the implementation. Persistent data identification in Texas has been associated with 
the instantiation mechanism, provided in the form of an overloaded operator \texttt{new}. Whilst the overloaded 
\texttt{new} is able to allocate any class of object, programmers cannot use the same operator to make their 
functions persist because these are not created through allocation routine calls.

To avoid the problem of dangling function pointers, Texas provides compile-time flags to 
generate code that sets the value of such pointers to \texttt{NULL} whenever they are encountered at commit time. This simple solution preserves the integrity of the store and has been used in some other projects e.g. 
Mneme’s Modula-3 [Moss 90]. Unfortunately, several objections can be raised and are listed below. Some of them have already been debated at length.

1. Setting such embedded pointers to \texttt{NULL} when the containing persistent object is committed to the 
store would cause the object to lose part of its closure. When the object is faulted back in during a 
later run, that information may no longer be present in the current execution address space.
2. The state of the object changes because of its persistent property. This is at best controversial.
3. The state change is carried out by the storage layer without programmer activity or permission. In 
fact, the programmer may well be unaware that this has happened. A subsequent dereference of this 
pointer is likely to cause abnormal program behaviour.
4. More convincingly perhaps, is the problem arising when the programmer has declared such a pointer 
to be constant. The storage layer would still set that pointer to \texttt{NULL} yet when the object is accessed 
in a later run, the pointer value though constant in the programmer’s eyes, has changed. Worse still, if 
type checking is enforced, the compiler would prevent any program from re-assigning a proper value 
to the pointer field. Essentially, that field becomes useless. In this case therefore, the semantics of 
constant references become unclear.

2.4.6. ObjectStore

ObjectStore [ODI 98] is one of the few commercial implementations of a persistent C++ system. Being a 
commercial product, ObjectStore comes with a rich set of language or library features. The design choices 
adopted bear similarities to those used in Exodus, Ode and Texas. Some notable features of ObjectStore 
are:

- Overloaded operator \texttt{new} for persistent objects. Similar to Ode but unlike E’s persistence, 
ObjectStore associates the persistence of its class instances with an overloaded operator \texttt{new}.
- Like Texas, ObjectStore maps in pages as and when persistent pointers are dereferenced for the first 
time. First dereference of a persistent pointer raises an exception that is trapped by ObjectStore, 
whose handler then brings in the page hosting the referred object. One difference is that whilst Texas 
will eagerly scan all contained pointers in that page and convert them to virtual memory pointers, with 
ObjectStore different implementation-dependent schemes are available.
- ObjectStore uses the term “persistency independent of type” to indicate that data objects of any type 
may persist without enforcing referential integrity and without supporting persistent functions. 
However, a discrimination exists between persistent and transient objects.
- Database roots that are somewhat similar to the Texas store approach yet restricted to point to 
instances of class extents only. This peculiarity is examined in greater detail below.
- A client-server architecture where clients are application programs and servers are ObjectStore 
database managers. As expected, clients run against the persistent store.
The ObjectStore persistent space is backed by several operating system files that contain the entire data space. As with PS-algol, each file is called a database. Files are in turn organised into segments and pages. Segments are logical regions of disk of arbitrary size that are made up of a number of fixed-sized pages. Pages are physical units of storage in that they are contiguous areas that ideally match the filesystem block size. In addition, clusters are a logical storage abstraction, visible at the programming level that can be used to improve performance. Segments and clusters allow the programmer to allocate objects in close physical proximity so that the application can benefit from locality of reference. At the storage level, ObjectStore's Virtual Memory Mapping Architecture makes use of memory mapping, clustering and caching to move pages between files and main memory.

A program first indicates its intention to use ObjectStore by making a call to the static member function `objectstore::initialize()`. Thereafter, a database can be created or an existing one opened using member functions from class `os_database`. For example,

```c
int main( int argc, char * argv[ ] )
{
    objectstore::initialize( ) ;  // Initialise the ObjectStore runtime system.
    // Create file /my_directory/my_data.db as a new ObjectStore database.
    os_database * db1 = os_database::create( "/my_directory/my_data.db" ) ;
    // Open an existing database.
    os_database * db2 = os_database::open( "/another/database.db", 1, 0644 ) ;
    // ...
    db1->close ( ) ;  // Closes the database referenced by db1.
    db2->destroy ( ) ;  // Discards the database referenced by db2.
}
```

The second argument to `os_database::open` is the opening mode: 1 is for read-only and 0 is for read-and-write. A call to `os_database::open` creates a new database if the specified name does not refer to an existing one. In this case, the third argument is the file protection mode. To prevent `os_database::open` from creating a new database, zero is passed as the third argument.

Unlike Exodus, Ode and Texas, the ObjectStore databases enforce type-security by properly containing run-time type information. ObjectStore maintains such class information, called schemas, in the form of C++ objects. However, ObjectStore requires the programmer to provide a schema file for each client application. This schema file must contain at least the top-level classes from which other classes identifiable from the top-levels may be tracked down. For example, given an application that handles persistent customer information, the file as shown in Figure 2-3 could be supplied to ObjectStore's schema generator by the programmer.
2. Persistence and Object-Oriented Programming

The schema generator will also include classes that, though not listed in the file, are identifiable from one of the classes in the list. For example, say class Product contained a data member of class Manufacturer, then the generator will also include Manufacturer. The generated information is then compiled and linked with the application.

ObjectStore programs have access to a number of overloaded new operators to create objects in a database. The two primary ones used for allocating single objects and arrays are:

```c
void * operator new ( size_t, os_database * dbptr, os_typespec * tptr );
void * operator new [] ( size_t, os_database * dbptr, os_typespec * tptr, int count );
```

and are used as in the following code fragment. Given a pointer db to an ObjectStore database, the two statements below allocate an integer and an array of 100 characters in the database. The integer is initialised to 9.

```c
int * iptr = new ( db, os_typespec :: get_int ( ) ) int ( 9 ); // size_t is implicit,
char * cptr = new ( db, os_typespec :: get_char ( ), 100 ) char [ 100 ]; // Ditto.
```

The os_typespec argument of the new operators is needed so ObjectStore knows what type of objects to create and what storage layout is needed. For all the primitive types, predefined static member functions of class os_typespec are provided. With classes, programmers are responsible for providing this information. ObjectStore provides two ways whereby programmers may achieve this:

1. by allocating os_typespec objects for every class for which there is an instance that needs to become persistent, or
2. by declaring a static member function called get_os_typespec in each class. The definition for that function is automatically generated by the ObjectStore schema generator. It returns a pointer to an os_typespec object.

To clarify the above, a simple scenario is used next: given the need to manipulate, say Customer objects, the following examples show how each approach is used. The following code fragment allocates and uses os_typespec objects. We note that os_typespec objects must be allocated on the transient heap.
os_typespec * Customer_type = new os_typespec ("Customer") ;
Customer * John = new (db, Customer_type) Customer ("John", "Citizen", "Old Kent Rd");

And the code fragment below uses the static member function get_os_typespec to achieve the same.

class Alien
{
    public:
    Alien (char * name) { // etc... } // An Alien constructor definition.
    static os_typespec * get_os_typespec() ; // Declare the type handling function.
private:
    char * name ; // Define a data member.
};

// Instantiate an Alien object. Note the need to call the type handling function so that
// ObjectStore can obtain the correct type information about the object.
Alien * idiot = new (db, Alien::get_os_typespec()) Alien("JarJarBinks");

So far, persistent object creation has been shown to be possible, essentially by specifying in which
database the object persists. Additional new and new[] operators are also available to give programmer
control over physical object locality within a database by specifying in which segment an object is to be
placed. Every database is created with two predefined segments—the schema segment and the default
segment. The schema segment is where all run-time type information is stored and is for internal use by
ObjectStore only. The default segment contains all objects allocated using the new and new[] operators
that we have examined so far. Additional segments can be created through calls to the member function
create_segment() of class os_database. Having done so, it is possible to allocate segment-specific
objects with operator void * new (size_t, os_segment *, os_typespec *) or with void * new [] (size_t,
os_segment *, os_typespec *, int) for arrays. For example:

os_database * db = os_database :: open ("database.db", /* etc... */); // Create a segment inside db.
os_segment * sg = db->create_segment();
int * iptr = new (sg, os_typespec :: get_int()) int(9); // Allocate an int in the segment.

The same applies for clusters, except that clusters are created inside segments. Also, unlike segments
clusters are of fixed size and may become full—in which case an exception will be thrown when more
objects are added to the cluster:

os_object_cluster * cls = sg->create_object_cluster (12000); // Create a cluster of
try
{
    char * cptr = new (cls, os_typespec :: get_int()) int(9); // Allocate an int
} // 12000 bytes.

// inside the cluster.
catch (err_objectstore e) { /* etc... */}
The overloaded new operators require either a pointer to a database, to a segment or to a cluster in which the new persistent object is to be allocated. ObjectStore also features a transient database in which transient objects can be created. A pointer to the transient database can be obtained via a call to os_database :: get_transient_database (), which can then be passed to the new operators. From existing literature, it is unclear what semantic difference if any, exists between ObjectStore’s transient database and its transient run-time heap.

In ObjectStore, deletion for both persistent and transient objects, is done via the default C++ delete or delete [ ] operators. Surprisingly, there are no overloaded forms of the delete operators for persistent objects.

Having examined how object creation and deletion is possible in ObjectStore, now there is also a need for programmers to access existing persistent objects from a database. To achieve that, ObjectStore allows programmers to define database roots as entry points into persistent databases—roots cannot be created in the transient database. Like Napier88’s persistence root that traditionally acts as entry point to an environment, an ObjectStore database root usually points to an extent, which is a collection of objects of the same class. Unlike Napier88’s environment though, an ObjectStore collection is similar to Ode’s type extent in that all contained objects must be of the same class. In version 2.0 of ObjectStore, collections are implemented as a library of templates that provide collections as sets, bags, lists or arrays. The following code fragment shows how a set extent can be rooted as a database entry-point.

```
os_collection :: initialize ( ) ;
os_database * db = os_database :: open ( "database.db", /* etc... */ ) ;
os_Set < Alien * > * Planet_Naboo ;  // Pointer to an extent.
Planet_Naboo = & os_Set < Alien * > :: create ( db ) ;  // Create the extent.
os_database_root * aroot = db -> create_root ( "Alien" ) ;  // Give the root a string name.
armot -> set_value ( Planet_Naboo ) ;  // Assign the extent value to
             // the root.
```

Henceforth, Alien objects can be placed inside the extent and later retrieved by accessing the root identified by string "Alien". The call to os_collection :: initialize ( ) is necessary before any use of the collections library is made by a program.

A database root is retrieved and its entry-point object accessed as shown below:

```
os_collection :: initialize ( ) ;
os_database * db = os_database :: open ( "database.db", /* etc... */ ) ;
os_Set < Alien * > * Planet_Naboo ;  // Pointer to an extent.
os_database_root * aroot = db -> find_root ( "Alien" ) ;  // Find the root given its name.
if ( aroot )
{
  Planet_Naboo = ( os_Set < Alien * > * ) aroot -> get_value ( ) ;  // Get the root object.
os_cursor < Alien * > c ( *Planet_Naboo ) ;  // Iterate over the set.
  for ( Alien * an_alien = c . first ( ) ; an_alien ; an_alien = c . next ( ) ) { /* etc... */ }
}
else cout << "Extent Alien not found" << endl ;
```
2. Persistence and Object-Oriented Programming

As would be obvious by now, ObjectStore provides a rich yet complex programming environment. ObjectStore has used an existing language, namely C++, in the hope that programmers will be spared from having to learn a new one. However, added complexity in the form of overloaded memory management operators, programmer supplied type information and object clustering policies have meant that ObjectStore presents nevertheless a fairly complicated programming environment to the software developer. Some of its features might actually be impediments that ought to be invisible at the programming level. In particular, the requirement that programmers manage type information is one such feature.

2.4.7. PJama

Despite being a language that appears to be more closely related to C++ than to PS-algol or Napier88, the Java programming language is an object-oriented language that fits more naturally in a persistence-by-reachability environment. The PJama [Atkinson et al 96, Jordan 96] project is a joint venture between Sun Microsystems Inc. and the Computing Science Department of Glasgow University. It is an experimental programming environment to support persistent Java. As was the case with Napier88, PJama follows the three principles of persistence independence, type orthogonality and persistence identification to yield orthogonal persistence. In addition, its design seeks to provide a system with the following properties:

1. Minimal changes to the language; ideally none.
2. No loss of safety compared to non-persistent Java.
3. A range of transaction models depending on programmer needs.

A PJama program executes on a virtual machine much in the same way as ordinary Java programs [Lindholm & Yellin 97]. A major difference is that the PJama virtual machine is backed by a reachability-based persistent store so that when the virtual machine starts up, it opens the persistent store and is thereafter ready to allow the executing program to interact with the store. A PJama program accesses the run-time store by making a call to static method `getStore()` of class `PJStoreImpl`. The call returns an object representing the current store. The code fragment below shows how such a call is made:

```java
import org.opj.store.PJStore;
import org.opj.store.PJStoreImpl;
// etc...
public static void main(String[] argv)
{
    PJStore pjs = PJStoreImpl.getStore(); // Get the current store instance.
    // etc...
}
```

Like PS-algol, PJama relies on "directories" of persistent roots through which objects can be made to persist. Unlike PS-algol however, PJama applications do not open stores—instead, the virtual machine is invoked with the store name as an argument. When the virtual machine starts, it may perform recovery actions if necessary and then it executes the `main` method of the class for which it was launched to run. Thereafter, objects can be added to the store either by making them reachable from an existing persistent object or by making root objects out of them through a call to method `newPRoot()` from the class `PJStoreImpl`. For example:
Object [] myObjects = new Object [15]; // Instantiate an array of 15 references.
for (int i = 0; i < myObjects.length; i++)
{
    // populate array myObjects here
}
// etc...
pjs.newPRoot ("Object Array", myObjects); // Create a new persistent root in the store
// and identify it with "Object Array."

Existing roots can be accessed using method getPRoot() as shown in the code fragment below:

import org.opj.store.PJStore;
import org.opj.store.PJStoreImpl;
// etc..
public static void main ( String [ ] argv )
{
    PJStore pjs = PJStoreImpl.getStore ( ) ;
    Object [ ] objectArray = ( Object [ ] ) pjs.getPRoot ("Object Array") ;
    if ( objectArray != null )
    {
        // Found it, so now we can manipulate objectArray just as though it was
        // created during this program execution.
    }
    else System.out.println ("Object Array not found in store") ;
}

In PJama, because most of the interaction between a program execution and the run-time store is dynamic, a number of exceptions may be thrown as a result of unexpected behaviour. In addition to the usual Java exceptions, PJama defines its own hierarchy of exceptions with PJException at the root. PJException itself extends Exception. So, when taking exceptions into account, all the above code fragments would have to be enclosed in try..catch blocks.

In its simplest form, support for transactions is implicit in that the start of main constitutes the start of a transaction. When main terminates, the transaction ends. Of course a program can explicitly end the transaction before main terminates. This is achieved either through a call to method stabilizeAll() or to method abortAll() of class PJStoreImpl. A new transaction start is implied immediately after the call returns. More elaborate transaction mechanisms are being explored in the PJama project and may be incorporated in future releases of the platform.

A number of implementations have been released [Sun 99a] since the PJama project started in 1996 and at the time of writing, there is on-going commitment from Sun Microsystems Inc. to continue the research endeavour.
2.4.8. Other Persistent OO Languages

So far this chapter has focused on research efforts associated with the popular OO languages. We now introduce a broader perspective on the subject at hand by looking at two additional contributions to persistence research. They are the Galileo and the Mozzie projects.

**Galileo**

Galileo [Albano et al 85] is a programming language that was developed for database applications. Some of the main features of Galileo are static typing, type hierarchies, abstraction mechanisms and persistent data. According to existing literature, Galileo attempts to address the semantic mismatch between the requirements of database development and the constructs available in common programming languages. Complex data structures available in high-level programming languages have no direct counterparts in most database languages. For example, heavily linked data structures or abstract data types usually cannot be represented in a database without a tedious mapping. This complicates storage and retrieval operations. Galileo represents a positive step to address the problem by integrating persistence into the language.

Galileo features a global environment in which data persists. Structures created in that global environment are simply stored permanently until destroyed explicitly. There is no distinction between the data types storable in the database and the data types of the language.

Having integrated persistence into the programming level, Galileo identified that there was a need to control names/value pairs in the persistent context. Galileo therefore features the *environment* construct, defined to be a mapping from identifiers to denotable values. Such an environment is obtained by evaluating an environment expression. The following example is borrowed from [Dearle 88]:

\[ \text{use } a := 3 \text{ and } b := 4 \text{ in } a + b \]

yields the value 7. Here, the expression

\[ a := 3 \text{ and } b := 4 \]

is an environment expression that evaluates to an environment containing the bindings, \( \{a,3,num,true\} \) and \( \{b,4,num,true\} \) in which the expression \( a + b \) may be evaluated. The example also illustrates two environment operations available in Galileo: the creation of new bindings using operator \( := \) and the combination of environments using \( \text{and} \). Galileo features other mechanisms to select a particular binding from a given environment, to recursively create names and values in an environment, and to remove a name from an environment.

As mentioned above, Galileo provides persistence through an environment called the global environment that always persists. The global environment may contain bindings including other “nested” environments. Every Galileo expression is evaluated with respect to an environment, initially the global one. The user may evaluate expressions with respect to another environment using the command *enter*. This command effectively allows the user to “traverse” the hierarchy of environments that is reachable from the global environment. For example:
use anenv := ( a := 3 and b := 4 )

! this defines an environment called anenv in the global environment

enter anenv:

! now the current environment is anenv

a + b

! yields 7 as before

The creators of the Galileo system suggest that environments could help the user to develop and test database schemata incrementally or to express the overall structure in terms of smaller related components.

**Mozzie**

The Mozzie Programming Language [Hollins et al 96] was developed at the University of Sydney as part of a research investigation that focused primarily on subtyping and encapsulation in persistent languages. The justification hinges on the premise that persistent systems may contain large volumes of data/objects that have been acquired over time by a large community of users. As such, there is a need to ensure that access to the data is properly controlled.

Mozzie is a prototype-based persistent OO language. It is strongly typed and its type system is mostly static and relies on structural equivalence. The type system features the types normally expected of a modern language: they are the scalar types, e.g. integers and strings, which display value semantics and objects, which display pointer semantics. Its object model is not too dissimilar from Modula-3’s object model [Cardelli et al 89].

In Mozzie, a new type definition can be specified by extending an existing type. The new type is known as a sub-type of the existing type. An important aspect of type/sub-type relationships in Mozzie is that a sub-type structurally "conforms" to the type it extends. This allows for variables of different types to refer to the same object. For example, if a type T is extended by a sub-type S, then given an object obj of type S, variables of both type T and type S may refer to obj. As we will shortly see, this has important implications in the way Mozzie supports persistence. With respect to persistence, Mozzie posits that the ability to cast objects from sub-type to super-type and back again, allows for a persistent store of objects that is built around the sub-typing rules of the language. Here is a simplified explanation of how Mozzie achieves that:

1. The Mozzie store is a reachability-based store with a single root, the value of which is well-known.
2. The type of the root of persistence is *Any*. This is the type of an object with no field and all Mozzie objects conform to this type.
3. From 2, it follows that any object can be assigned to the root and the object therefore persists by virtue of its reachability from the root. To retrieve the object, the root is dynamically cast to revert its value back to its original type.

We note that Mozzie’s dynamic type projection capabilities are all type-safe—any invalid projection causes an exception to be raised.

Another major contribution of Mozzie has been to show that a single model of protection is possible in a persistent environment. The language uses capabilities to control object access. In fact, all object references are capabilities so that every object in the Mozzie persistent space is implicitly associated with at least one set of access permissions that regulates its use for as long as the object
persists. An object may have many capabilities each of which potentially specifies a different set of permissions on that object. Then, a program that accesses that object will find it can only perform those operations that are allowed by the capability it holds. Obtaining a different capability will yield an altogether different set of rights again.

2.5. Summary

This chapter has introduced background material on the persistence abstraction, in particular orthogonal persistence, and its related issues. The approaches adopted by various research efforts world-wide have also been examined and some strengths and weaknesses were identified. Certain systems have been singled out for investigation because they are relevant to the object-oriented setting that underlies this thesis. Also, they present peculiarities that have contributed towards understanding how to approach persistent systems design. First and foremost, from the approaches used by the various projects examined, two main philosophies of persistent data identification have emerged. They are the reachability and the allocation-based approaches. The experience with PS-algol, Napier88 and PJama strongly indicate that reachability is a pre-condition for orthogonal persistence. Additionally, a study of O++ also revealed one way to support object versioning. Such facilities are absent in the other projects that were examined. And finally, two noteworthy observations are:

1. Most project investigators have found some clustering technique, whether logical or physical, to be desirable. This is evidenced in Napier88's environment, the clusters of O++ and ObjectStore's segments.

2. Persistent type information is important. Some of the implementations examined, namely Texas and ObjectStore, make it a programmer duty to either provide type schemas to the store or to specify type names in function/macro calls when interacting with the store. With PJama, orthogonal persistence ensures that the class information of any persistent object also persists.
Chapter 3
Object-Oriented Orthogonally Persistent Languages

3.1. Introduction
The preceding chapter introduced the persistence abstraction and in particular, it focussed on several systems that support persistence in a few object-oriented, OO, languages. If there is a flavour-of-the-decade in programming languages, for the 1990s it would have to be OO languages. It is therefore not surprising that with the widening acceptance of object-orientation, researchers were quick to develop persistent object-oriented systems such as those reviewed earlier.

The popularity of the OO programming model is understandable in the light of the following observations:

- OO programming is a normal step forward from data abstraction and encapsulation.
- Many real-life entities are objects and a straightforward one-to-one correspondence can usually be found between a real-life entity and the object representing it in a program. In other words, object-orientation has been found to model the real world in a way that is natural and as close to the programmer’s perspective as possible. Strictly speaking, the term object-orientation here describes a design methodology, as opposed to language features. However, such a design specifies the structure of an application in terms of objects and relationships between objects, and frequently, implementation is eventually carried out in a language supporting OO programming.
- OO programming encourages code reuse and extension through certain features such as inheritance and sub-typing.
- In non-military development environments, C++ and Java are major language choices by software engineers. C++ also appears to be increasingly chosen for military applications development in the light of the U.S. Defence Department’s decision to sometimes allow contractors the freedom to use a language other than Ada. More recently, Java has gained increasingly wide acceptance especially in the areas related to network programming.

The first part of this chapter reviews the fundamentals of the object-oriented paradigm. Apart from the most basic concepts such as objects and methods, there seems to be little consensus as to what exactly is object-oriented and what is not. To that effect, we now discuss those object-oriented features deemed essential for the discussion in this thesis. The literature seems to indicate that there are at least two kinds of OO languages: the so-called class-based languages exemplified by C++ and Java, and the prototype-based ones such as Obliq [Cardelli 94]. In this thesis, the emphasis is on class-based OO languages where objects are created through class instantiation.
3. Object-Oriented Orthogonally Persistent Languages

3.2. Relevant Object-Oriented Concepts

Some object-oriented purists have commented that the question, "is language X object-oriented?" should not be asked. Instead it is more meaningful to ask, "to what extent is language X object-oriented?" There is however a need here to identify those language features that are deemed to be supportive of OO programming. If anything, this exercise in classification adds focus to the discussion in this thesis. Therefore for the purposes of this thesis, the following are primary concepts that are expected to be present in class-based OO languages:

1. First and foremost, all OO languages support the concept of an object, which is essentially a unit of encapsulation of state and of behaviour, that can be referenced using constructs available in the language. Chapter 5 will provide more details on the object concept adopted in this thesis, including semantics suitable for a distributed setting.
2. Object creation through some class/type instantiation mechanism. In other words, objects are instances of classes.
3. Object identity. To be able to reference objects, there is a need for some kind of unique and immutable handle on objects. The same is needed for differentiating objects. Object identity is an intrinsic property and is independent of class or state.
4. Support for abstraction. By this we mean the ability to identify the behaviour of an object that distinguishes it from objects of other types. An abstraction mechanism allows an object to present a well-defined behaviour to outsiders.
5. Encapsulation—the ability to self-contain and to hide the main attributes of an object from intrusion. Coupled with abstraction, encapsulation affords the ability to present an external view to an object’s behaviour without permitting outsiders the ability to peek into the object’s internal state.
6. Run-time type identification. This enables objects to know which class they belong to, so that at any point in an object’s lifetime it may query its run-time type identity.
7. Inheritance. The ability to define new classes by extending one or more existing classes.
8. Sub-typing. Sub-typing allows for objects of a certain class to be used in a situation where another class was expected, and this without breaking type security rules.

The subjects of sub-typing and inheritance are sufficiently important to warrant more detail. Inheritance is an object-oriented feature that allows new objects to be defined through the re-use of existing class definitions. It essentially organises objects into hierarchies of classes such that an object lower down the hierarchy can be implemented as an “increment” of one that is higher up the same hierarchy. Typically such re-use also involves the addition of state information and the addition or overriding of behaviour/method specification.

With respect to sub-typing, a relationship, called an is-a relationship must exist between a type and its sub-type. A sub-type is a type that is derived from and represents a specialised form of the original type. OO languages provide a mechanism for sub-typing so that if A is a sub-type of B, then any expression evaluating to a result of type A may be used where an expression of type B is expected without jeopardising type-safety. In other words, the primary aim of sub-typing is to support the ability to substitute an expression for another in a type-secure way. Inheritance and sub-typing are two separate concepts although they are sometimes supported through a single mechanism as section 3.2.1 will examine.
Closely associated with sub-typing, is dynamic binding. The observed behaviour of an object depends on its run-time type identity and therefore, it is at the time of invoking a method on an object that the correct method is identified.

Method override/overload is also commonly practiced in conjunction with sub-typing. This involves the re-definition of methods and/or addition of semantics to existing ones and results in behaviour that is more specific to the sub-type produced.

### 3.2.1. Some Popular OO Languages

This section briefly looks at three languages that support object-oriented programming, namely C++, Modula-3 and Java. In their own ways, all three of them support the concepts that were deemed important, as was discussed in the preceding paragraphs. With respect to C++ and Java, the term “type” is used to mean a super-set including “class” definitions and primitive types. The term “class” is not part of Modula-3 terminology. For detailed aspects of these languages, the reader is invited to consult [Stroustrup 93, Cardelli et al 89 and Gosling et al 96].

**The C++ Language**

C++ [Stroustrup 93] started as “C with classes” but was soon revised to include more OO features. In addition to the C facilities, the relevant features of C++ are classes, inheritance and sub-typing, reference variables, operator and function name overload and run-time type identification. We note that C++ combines inheritance and sub-typing into a single mechanism so that if a class A publicly inherits from class B then A is a sub-type of B.

The key concept to OO programming in C++ is the class, which is a user-defined type through which objects can be created. A class is a blueprint for the objects it can instantiate, either on the run-time stack or in heap memory through a call to operator `new`. It specifies the data layout of, and the operations defined on objects of the class. All objects have a unique identity, accessible through the implicit `this` pointer that is a property of each object. For objects instantiated through a call to `new`, a corresponding `delete` operation releases the memory used by the object.

A class also defines member functions to regulate the behaviour of its objects. Member functions are functions that are meant to operate on objects of the class. Methods are defined before run-time, typically at the time of creating the class definition. Additionally, most operators, e.g. the arithmetic and the boolean ones, can also be re-defined by a class.

Encapsulation is afforded by way of access “regions” within the C++ class definition. Members can be declared within one of three regions—private, protected or public. Data members, also known as state variables, that are private are completely encapsulated whereas the protected ones are accessible to derived classes. Data members can also be “instance” variables or “class” variables—every object of the class has its own copy of an instance variable whilst a class variable is a variable/object in its own right, globally shared by all objects of the class.

As mentioned above, C++ mixes inheritance and sub-typing into one derivation mechanism. In other words, C++ does not provide constructs that separate the two concepts—when a class sub-types from a super class, the former inherits all the behaviour of the latter. C++ supports multiple inheritance and allows a class to inherit from more than one base class.

Recently C++ has undergone revision and the ANSI/ISO standard definition of the language as it stands at the time of writing this thesis, is the X3J16/96-0225 WG21/N1043 C++ Draft Standard. In it,
provision is made for the support of run-time type identification although actual implementations differ widely in that respect. As a minimum, the Standard requires an object of class type_info—which is a type information object—to be associated with every class for which an instance exists at run-time. Operator typeid is provided to discover run-time types. Given a type name as its operand, typeid returns a reference to a type_info object that represents the type name, and given an expression as its operand, typeid returns a reference to a type_info object that represents the type of the object denoted by the expression.

Modula-3

Modula-3 [Cardelli et al 89] is a general purpose language of the Pascal family, that supports OO programming. It offers roughly the same facilities as C++ except that whilst power and flexibility drive the development of C++, in Modula-3 safety and clarity are the main concerns. Modula-3 provides the usual primitive types, first-class procedures, records, heap-based objects and their methods and, sub-typing and single inheritance in a single mechanism. In addition, the run-time system incorporates automatic garbage-collection. Pointers are called references and are declared by preceding the type being pointed to, with keyword REF, e.g. REF INTEGER.

An object in Modula-3 is either NIL or it is a reference to a data record coupled with a method suite. That is, an object is similar to a REF RECORD except that additional fields, i.e. inherited ones, may be part of the data layout. The method suite is also a record—but one of procedures, each of which accepts as a first argument an object of the type or of a super-type.

Modula-3 objects are allocated on the heap and are always manipulated through references. They must be NEW’d—the call to NEW expects the type name as a first parameter and creates an object of that type. A Modula-3 object includes a typecode that can be tested to determine its type dynamically.

Modula-3 programs are made up of a number of modules and interfaces that are separately compiled and linked together into an executable program. Interfaces are collections of type, variable and procedure declarations that reveal the public part of a module. For every interface there is usually an associated implementation module that at least defines the procedures declared in the interface. The module is said to implement the interface. Anything else defined in the module is private by virtue of its absence from the interface. A module imports interfaces it depends on and exports the interface(s) that it implements.

Modula-3 promotes the superiority of structural equivalence whilst acknowledging that name equivalence is sometimes necessary. To that effect, Modula-3 decides type equivalence on structure but offers the ability to “brand” a type to inform the compiler that name equivalence should apply on instances of that type.

Garbage collection is an integral part of the Modula-3 run-time system. As mentioned above, dynamic memory usage is through the built-in operator NEW. When there are no more references to a dynamic variable, it is automatically garbage-collected.

Like C++, Modula-3 supports inheritance through the sub-typing mechanism; unlike C++, only single inheritance is supported. A new object type may be defined as a sub-type of an existing type, in which case objects of the new type inherit all the data fields and methods of the old type and possibly define some new ones too. The sub-type may also provide new implementations that override selected methods of the super-type. In addition to such “definition-time” overrides, method override may also happen at object instantiation time, creating new anonymous types at the same time. By default, all new types derive from the predefined type ROOT, which is the base of the Modula-3 inheritance hierarchy.
This brief treatment on Modula-3 has been included here because as shown in the next section, some important Modula-3 features—for example its object model—have been adopted into Java.

**Java**

The Java Programming Language [Gosling et al 96] is fast gaining popularity and importance as an application development language. It is particularly becoming the language of choice in the web-based and distributed computing environments. Indeed, the compile-once-run-anywhere gospel preached by Sun Microsystems Inc. has been attractive to many. Judging by the number of persistent Java projects that have been initiated over the past few years, it is likely that the persistence research community is relying on Java to popularise the persistence abstraction in the wider programming community. Unlike C++ and Modula-3, Java is almost pure object-oriented—primitive values are not objects although they can be wrapped into one. It was created as a primarily OO language and the following features are relevant to the discussion in this thesis.

Java’s basic unit of programming is the *class*—all code must belong to some class. Java’s class construct is closer to C++’s class than it is to Modula-3’s type definitions. However, Java’s object model is closer to the Modula-3 model and the Java object is a reference, made to refer to a storage region—the actual object—through a call to `new` or initialised to `null`. The class specifies the fields that an object will have as well as defines the methods that implement object behaviour. Each field and each method of a class has an associated access control specifier in the form of keywords `private`, `protected` and `public`, which have similar semantics to the C++ access regions. In addition, Java features `package` visibility as discussed below. The degree of encapsulation depends on the programmer’s use of these specifiers.

Java supports the concept of a package of classes to help programmers avoid name clashes by organising their code into logical units. Packages are “sets” of classes and sub-packages; it is a programmer’s responsibility to give meaningful semantics to their package organisations. Associated with the package concept, is the `package` visibility. Class fields and methods declared with no access specifier are of `package` visibility.

With respect to inheritance, again Java is closer to Modula-3’s single inheritance approach in that a class can only extend one base class. However Java features an `interface` type [Fisher 96] and a class may implement any number of such interfaces. Like Modula-3 therefore whether the programmer makes it explicit or not, all Java classes extend class `Object`, the base of all Java classes.

Garbage collection is included in the run-time system. Unlike C++ where objects need to be explicitly deleted, Java reclaims memory using reachability criteria, i.e. any object that can no longer be reached by a reference is deemed reclaimable.

Java provides extensive run-time type identification and manipulation support. Much like the `type_info` objects of C++, Java type identification is supported through class `class` objects associated with every class and interface present in the run-time system. Operations defined on class `Class` objects are however more extensive and complete. Run-time type manipulation is supported through reflection, implemented as package `java.lang.reflect`.

### 3.3. A Model of Object-Oriented Orthogonal Persistence

Having examined relevant OO concepts, a model of orthogonal persistence for class-based object-oriented languages may now be proposed. The model developed here draws heavily from existing technology that
has proven viable in supporting the Napier88 Persistent Programming Language. In particular, the Napier88 technology is found to be sufficiently generic to apply to OO languages. As discussed in section 3.2, OO languages display some interesting properties or concepts that are of direct significance in an orthogonally persistent environment. In addition the following comments are noteworthy:

- Persistent type identity. Whilst section 3.2 included run-time type identification as a relevant concept, strictly speaking, the run-time context is only adequate in a non-persistent environment. This is because in such an environment, object lifetime is delimited by execution time and hence type identification is only needed for as long as the execution lasts. In a persistent environment, the type identification needs to be as long-lived as the objects needing identification. It therefore follows that type information needs to persist. This observation was first made in [Atkinson & Buneman 87].
- Orthogonal persistence. As chapter 2 examined, orthogonal persistence was seen to be a most versatile form of persistence that avoids problems such as dangling references.
- Storage hierarchies and management. Objects may outlive program execution and be retained in a persistent store. Automatic garbage collection becomes important to ensure store integrity.

The next section expands the model of object-oriented orthogonal persistence that is the subject of this chapter. For more concreteness, an actual design and a prototype implementation is developed. C++ has been chosen as the vehicle for experimentation because it has emerged as a major programming language. C++ has also been perceived to be a "difficult" language when it comes to adding persistence.

3.4. Orthogonally Persistent C++

The previous chapter examined various endeavours at implementing persistent C++ programming systems. All of them opted for allocation-based persistence, thereby extending the programmer's responsibility for dynamic memory management into the persistent space. Consequently, these systems displayed the problematic behaviour associated with manual deallocation of memory but exacerbated in a persistent environment. Also, additional problems arising from a failure to support orthogonal persistence were introduced.

World-wide research efforts in reachability stores have matured over the past decade and there is now common understanding on design approaches for building reachability persistent stores [Brown & Morrison 92]. These stores have a sufficiently generic architecture to support an orthogonally persistent implementation of C++ on them. This section documents an experiment conducted for the purpose of showing how orthogonal persistence can be supported in C++. Aspects of this experiment were published in [Lew & Brown 97, Lew & Brown 98]. A prototype implementation is used as an example throughout this section. Since C++ was not designed as a persistent language, a number of language issues need to be addressed. They are first discussed and possible solutions are suggested.

3.4.1. Primary Issues Arising in C++

In supporting orthogonal persistence in C++, there is a need to consider the following issues:

1. Any data object that may be referenced—whether of primitive, class, or union type—may persist. Here class includes an object of struct type.
2. Functions should be able to persist. To determine this requirement, we consider some persistent object containing a pointer to some function. When the object is written to the persistent store, the function must be retained in order to avoid a dangling reference.
3. Member functions should also persist. Just as objects may reference functions, a persistent object may also contain a pointer-to-member-function referencing some member function. The importance of this requirement however, is that virtual functions support is facilitated.

4. Individual data members could also persist. Again, a persistent object may contain a pointer that references into an object.

The above four issues, especially 2 and 3, call for persistence by reachability.

**Persistence of Functions**

A function should be retained in the persistent store if a pointer to the said function becomes reachable from the store root. We support persistent functions in the way we introduce compilation units in the store. Each unit produced by the compiler is a shared object file that becomes a code object. Associated with the code object is a number of names representing functions that have their code inside the code object. The code object is reachable from these names and can be dynamically mapped in. The mapping function returns a pointer to a function and thereafter a dereference of the pointer calls the mapped function.

**Persistent Unions and their Semantics**

Unions are objects that may, at different times, hold data of different types. In a non-persistent environment, when a union object is created, the creator program is assumed to have the necessary type information to correctly access the information within. This assumption breaks down in a persistent environment where data outlives program execution and is thereafter accessible by other programs. We associate a tag with each union to keep track of the active member and it is a compiler responsibility to generate code to update the tag every time the member changes.

Unfortunately, the compiler may not always be aware of active member changes. The use of references is a prime example, as shown in the code fragment of Figure 3-1. We solve this by making the compiler keep track of which references or pointers access unions, although we do note that implementing such a smart compiler is a tedious undertaking.

```cpp
union u_type {
    float a_float ;
    int an_int ;
} u ;

u.a_float = 3.1415 ;  // Active member is a_float.
int & iref = u.an_int ; // Declare a reference to u.an_int.
iref = 10 ;           // Now change active member via the reference.
```

**Figure 3-1:** Aliasing a union member.

**Garbage Collection**

C++ memory management is very much under programmer responsibility. This is unfortunate since such explicit control can be unsafe in the persistence context. Three potential errors sources are (i) dangling pointers when an object referenced by more than one pointer is deleted, (ii) deleting an array with `delete`, (iii) deleting a single object with `delete[]`.
Situation 1 appears inevitable, it is a consequence of leaving memory cleanup to programmers. In case 2 and 3, it is unfortunate that the C++ Reference Manual does not regulate on the result. In 2, the destructor is called an incorrect number of times and the wrong memory size is freed, in 3, the destructor may be called an incorrect number of times and memory beyond the object is deleted. In both cases, behaviour is undefined. To address this, we incorporate garbage collection and redefine the semantics of delete or delete [] to avoid conflict with garbage collection. The following happens upon deletion:

1. The destructor is called and the lvalue expression or the pointer denoting the object in the delete expression is set to null.
2. The storage occupied by the object is eventually swept away during garbage collection if there are no references to it.
3. If a "deleted" object is accessible from a different expression or pointer, then evaluating the expression or de-referencing the pointer still accesses the object. However, the state of the object might have been altered. Deleting the object again calls the destructor once more.

These semantics do not contravene the Language Reference Manual. In any case garbage collection only reclaims memory and does not call destructors. This is because the destructor should only be called upon object deletion. This happens when one of the delete operators is applied to a heap object or when a stack object goes out of scope. In the latter case, the object is an automatic one and no memory de-allocation is needed, in the former the pointer is set to null. In any other case, the object is conceptually always present, but inaccessible. The rationale for such semantics is that reachability identifies data as persistent if they are reachable. In graph structures where some nodes are accessible via more than one computation expression, a node is only discarded when no edge leads to it.

Although storage space may not be immediately reclaimed after a delete operation, there is still the need for the destructor to be called—the programmer expects the destructor to be called. And since persistent objects may be accessed repeatedly by many program invocations, every time an object is deleted or goes out of scope, the destructor will be called.

[Stroustrup 93] identifies two types of garbage collectors that could be used in C++:

1. Copying collectors that move objects to compact fragmented space,
2. Conservative collectors that allocate objects in a way that minimises fragmentation.

Stroustrup also prefers conservative collectors over copying ones because it would be impractical to move an object and then modify all pointers to it in the program referencing the object. However, this is possible using a pointer stack.

**Run-Time Type Identification, RTTI**

The 1995 ANSI/ISO C++ standard requires type information to be accessible at run-time. In non-persistent systems, this meant that type information needed to be incorporated into every executable. This would usually be in the form of type_info objects associated with every class. Then at runtime, any class object may query its type_info object. However, where object longevity may exceed program execution, type information availability should be dissociated from execution. In our model, type information is persistent. We make it a compiler task to build a graph representation of every class definition that is compiled and to introduce it into the persistent space. The compiler also introduces the executable code for member functions in the form of code objects which are reachable from the graph representation of the class. Once a class definition is in store, any main program that needs to refer to it would "include" it. The
main program is also compiled into the store and thereafter, the linking stage assembles an executable representation of the program. An executable is simply a mechanism allowing the necessary code objects to be accessed for mapping in; it is the set of all possible links that an execution may traverse in the course of computation.

**Persistence and the Storage Class static**

We now discuss the impact of persistence on the storage class *static*. Whilst the syntax and context in which static is used remains unchanged, the meaning is totally different. This is especially important for static class data members.

**Global and local static objects**

In the C++ Language Reference Manual, an object means a variable or an ordinary function. A class object means an instance of a class. The same terms are used here.

Program start is defined as the execution of `main()`. Termination occurs either when `main` executes a `return`, goes out of scope, or the program calls standard library function `exit()`. At the time `main` starts, the language requires all global objects to have been initialised, either to a programmer-specified value or to zero, converted to the appropriate type. These semantics need to be retained despite the blurring of program boundaries in a persistent environment.

Two meanings could apply to objects with static values. They are explained next.

1. Variables of global scope inside a compilation unit are initialised before their first use and retain their values thereafter. Therefore the declaration of a global variable could mean that it is *implicitly persistent*. Once created and initialised, it is forever retained and programs would not re-initialise it by default but would see the latest value. This approach is appropriate when say, a global counter is used to generate primary keys for the tuples of a database relation.

   Strictly speaking, this meaning is actually what would be expected of variables with static values when persistence is introduced. The Language Reference Manual requires all static values to be retained till the next time they are in scope. Therefore, any static counter would keep on incrementing. However, this is not always desirable.

2. The alternative is to give each run a fresh copy of the variable. Since global variables may be referred to by other programs after the original one has exited, this may lead to surprises if initialisation is expected but not performed. A fresh copy is necessary when an execution is not dependent on previous runs. For example, a restaurant patron must not have to pay for previously settled meals.

Whilst the first meaning is closer to the Reference Manual, it can be impractical. If this were the only option, then most programs that make use of static values would work in the first run only. There is a need to support both meanings. One solution is for the compiler to generate appropriate code, given a compile-time flag. This method has the advantage of not requiring any source level modification of existing applications when compiled into the store. The disadvantage is that from the source code, it is impossible to tell which global variables are implicitly persistent and which ones are not. The syntax doesn't differentiate between them, so the programmer must document it. Also, a compiler flag is very coarse-grained in that the flag would apply to a whole compilation unit. Either all the globals are compiled as implicitly persistent or they are not. It would not be possible to have say, a persistent global counter for invoice numbers and a transient global for customer charge in the same compilation unit.
Static class members

Specifier static, when applied to a data member of a class, indicates there is only one instance of the member. That member is not part of any particular object of the class but is shared by all. At the time of introducing a class into the store, the compiler could define all the static data members. These will be individual objects within the representation of the class. This ensures that these data members are independent of any class object and in fact, exist even if none is in store. There is a need to retain the usual semantics of data members declared static yet this approach makes such data members implicitly persistent. Again a starting program may need to be independent of any previous values of a static data member. A fresh copy of the data member is needed. It follows that a major problem here is to decide the type equivalence of class objects not sharing the same static data member.

For a program to run without worrying about side-effects of other programs executed before the current run, it needs a copy of all such data members. Unfortunately this contravenes the requirement that there be only one instance of such data members, unless they are considered to be of two different types. Treating class objects having private copies of a static data member, as different types, solves the problem. However, every time such a program runs, objects of those clone types could populate the store and become useless. On the other hand, if those objects are considered to be of the same class, an ambiguity could arise as to which static data member applies when a later program accesses several of these objects.

Two possible solutions are: (i) treat all static data members as implicitly persistent and use a global static variable for the other case if it is needed, (ii) let static data members be transient. Of the two, (i) is preferable because it is closer in meaning to the Language Reference Manual, although existing traditional applications may not run correctly in this persistent model. A fix can be provided in the form of a utility program to set a static data member to a desired value before the program executes.

So, given the code fragments as shown in Figure 3-2a and b, solution (i) means executables would share the static data member T::count but would each own a separate copy of the variable count declared in T.C. Solution (ii) means that an instance of the class that outlives its creating program cannot assume anything about the value of any static data member the next time the instance is in scope. This is unacceptable.

```
// file T.h
class T
{
    static int count; // Declaration of a static data member.
    char name [20]; // Definition of a non-static data member.
public:
    T();
};
```

Figure 3-2a: Static data member declaration.
Persistence and operator delete

In traditional C++ models, the deletion of a heap-based object calls the destructor, if any, and frees memory. However, in a persistent environment, an object should only be swept away by the garbage collector. The Reference Manual states:

- the delete operator destroys an object created by the new operator and the effect of applying delete to a pointer not obtained from the new operator is undefined,
- the effect of accessing a deleted object is undefined,
- if the expression denoting the object in a delete expression is a modifiable lvalue, its value is undefined after the deletion,
- the delete operator will invoke the destructor, if any, for the object pointed to.

There is a need to uncouple the two activities that happen when an object is deleted. In our orthogonally persistent model, the following happens when an object is deleted:

1. The destructor is called and the lvalue expression or the pointer denoting the object in the delete expression is set to null.
2. The storage occupied by the object is not immediately freed. This happens at a later stage when the object has no more references to it and is eventually swept away by the garbage collector.
3. If a “deleted” object is still accessible from a different expression or pointer, then evaluating the expression or dereferencing the pointer will still access the object. However, the state of the object might have been affected by a previous call to the destructor. Deleting the object through the second pointer calls the destructor again.

The rationale for such semantics is that reachability-based persistence identifies data to be persistent if they are reachable from one or more roots [Brown 88]. In graph structures where some nodes are accessible via more than one computation expression, a node should only be garbage collected when no graph edge leads to it. Retaining “deleted” objects until they are unreachable also ensures there are no dangling references in the store.

Concerning arrays, the Reference Manual states:

- the form delete [ ] cast-expression is used to delete arrays,
- the effect of deleting an array with the plain delete is undefined, as is deleting an individual object with the delete [ ] syntax.

In general, not using the appropriate form causes havoc at run-time. This is because in most existing implementations, delete [ ] relies on the existence of a header at the start of arrays. The header stores the number of objects in the array and by accessing this information enables the destructor to be called the correct number of times. However, single objects either do not have such a header or the header has a
different format. When a single object is deleted with \texttt{delete [ ]}, this causes the destructor to be called an incorrect number of times and to potentially access memory that is beyond the bounds of the array. Likewise, if an array is deleted with the plain \texttt{delete}, the destructor is called only once—for the first array element. Additionally, in both cases an incorrect memory size is freed.

In our model, because all objects carry a description of their shape and size, even if the programmer uses the wrong delete form, the de-allocation function implementing the delete operator will call the destructor the correct number of times. Also, the garbage collector will eventually free the right space for all unreachable objects.

3.4.2. The Program Development Model

The translation units of free-standing C++ programming systems have traditionally been filesystem-based. In the Language Reference Manual context free-standing means non-embedded. That is, source code is stored in individual files, the compiler generates object code files from the source code, and finally the linker assembles the object code into an executable file. Essentially, the linker assembles the executable by copying the relevant parts of the object code files into the executable file and resolving the references among the various parts of the object code.

The diagram in Figure 3-3 illustrates the traditional model of C++. Although this illustration shows linking to be an obvious separate step in generating the executable, this need not always be so.

![Figure 3-3: Traditional compile-link process in a file-based model.](image)

In practice, this may generate erroneous executables because the linkage is not type-safe. For example, existing C++ compilers that rely on the Unix system linker, do not detect the following error when the two files shown in Figure 3-4 are separately compiled and their object code files are linked together:
In the code above, an object `point` is defined as a coordinate of two dimensions. However, when accessed from `main()` where a three-dimensional coordinate class definition is in scope, it is wrongly linked to the methods of the local class.

In a programming environment backed by a persistent store the above situation can be easily detected. In such an environment, class definitions would be compiled as stand-alone units and then introduced into a user-specified scope of the store. Once in store, any main program code that needs to refer to those definitions would "include" them. The main program would also be compiled into the store and thereafter, the linking stage would assemble together an executable representation of the program. This is shown in Figure 3-5.
In this model, executables are truly able to share code instead of being each given a private copy. An executable is simply a mechanism to allow the necessary code objects to be tracked down at run-time. It is the set of all possible links that an execution may traverse in the course of its computation. An execution is a particular run of an executable.

The use of a persistent store to retain knowledge of user-defined types encountered by the compiler has several advantages. Two main ones are:

1. Persistence of types enables all compilation units to refer to one single representation of a class which has already been seen by the compiler,
2. It is possible to enforce name-based type equivalence throughout an entire application.

In view of the 1995 draft of the ANSI/ISO C++ standard, persistent types also ease the implementation of the proposed casting operators. Non-persistent C++ systems have to maintain run-time type information, RTTI. This information would be readily available in an orthogonally persistent C++.

In this project, persistence is by reachability and a generic persistent store architecture [Brown & Morrison 92, Brown et al 92a] is used to provide the required persistent address space. Conceptually, the space is organised into three main partitions:

1. a type namespace in which all user-defined type representations and their methods can be in scope,
2. an area for the representation of primitive types, and,
3. a partition for objects, namely class objects, executables and other variables that have outlived program execution. This area is accessible from the conceptual root of persistence, PS.

The primitive types area is static and the information is encoded at the time of creating the Napier88 store file. The type and the object spaces are dynamic and may be further divided into smaller spaces using containers similar to the Napier88 environment concept [Dearle 89].

**Compiling user-defined types into the store**

For programs to share persistent objects of a certain type in the store, they will need to know about that type. This is necessary so the programs know how to manipulate those objects. Such type representations, obtained from C++ class definitions, are introduced into the persistent type-space by the compiler since this is when all information about the class is encountered for the first time. Typically, this would be done at the time when the class definition is compiled into the store:

```
> compile T.C include_env
```

The above command would compile T.C and place a representation of class T into a “container” type-scope called include_env. Having different containers, each holding a number of classes, enables different versions of a class to co-exist. Containers effectively partition the type space.

**Binding style at link time**

Given the above executable format, it is possible to support both static and dynamic binding of code. Consider the program in Figure 3-6, that creates objects of some classes.

```
#include <iostream>
#include "T.h"
#include "Complex.h"

int main ( )
{
    T * tp = new T;
    Complex z ( 1.1, 3.9 ) ;
    z.print ( ) ; // Call Complex member function print ( ).
    tp->do_something ( ) ; // Call some function from class T.
    // etc...
}
```

*Figure 3-6: Sample program.*

After all the necessary compilation units are in store, they need to be linked. Binding can now be dynamic or static. For example, say `main ( )` wanted to always use the latest version of class `Complex` but only use a specific version of `T`. The linker could then be instructed to do so:

```
> link PS/code_env/myprog -sb include_env/T -db my_env/Complex -o PS/bin
```
The above command links the object code and outputs the executable `myprog`. `myprog` is created inside container `bin`, reachable from `PS`. The flags `-sb` and `-db` specify static or dynamic binding. Thereafter, should new versions of `T` and `Complex` be compiled into the store to “replace” the existing representations, `myprog` would see the latest `Complex` class yet retain knowledge of the same `T`.

The rationale for choosing static or dynamic binding depends on the development environment which the programmer operates in. There is a strong case for dynamic binding when the application is still under construction. It is common that constituent compilation units are constantly being refined and improved upon. With dynamic binding, a change made to a compilation unit will be reflected in all its dependencies, without recompiling them. Once the application is ready, the final linking can then be static. This avoids affecting the current version in the event work on a new release starts sometime in the future.

With dynamic binding, type security becomes an important issue. Existing objects instantiated under a former definition of a class `T` may no longer be compatible with a new `T`. To avoid “breaking” existing objects, at the time of recompilation, the compiler should check the structural equivalence of `T` with any existing representation, if present. For the purpose of this check, the “structure” of a class is defined to be its declaration, which would usually be in a header file. Only if the two structural forms agree, may the new representation take the place of the existing one. An extensive treatment of type equivalence checking is given in [Connor 90].

The decision to define class structural equivalence on the whole declaration as opposed to only the data members is based on the following reasoning: whilst class evolution would be more flexible if it is possible to add member functions, there is a need to ensure that two classes aren’t deemed equivalent merely because they have the same data layout. A safe and simple way to augment this simplistic approach is to include member function declarations into the equivalence test. After all, object-oriented technology has always promoted the object concept as being more than a mere data structure but as a combination of data members and associated methods.

Trouble arises when the declaration of `T` is unchanged but the semantics of the data members have changed, and hence the operations of member functions have different meanings. Existing objects will still type-check correctly and will be able to call the new member functions but erroneous results are obtained. Unfortunately it is not possible for the compiler to detect such changes because the types do match. It is then the programmer’s duty to avoid such situations. The availability of dynamic binding does not relieve the programmer from having to correctly interpret the semantics of persistent data. Semantic misinterpretation is not a problem arising from introducing persistence into a programming model. Even in the traditional file-based model the same difficulty, if not worse, arises when objects are written out to a file and are subsequently read in during a later run.

Finally, dynamic binding ideally offers the flexibility of changing parts of an application while the application is still running. Here again it is the programmer’s responsibility to ensure the semantic integrity of the data. The compiler only ensures type security.

**Linking compilation units into an executable.**

When the compilation unit containing the code for `main()` is statically bound to other compilation units, the binding is to the actual code object in store. In the diagram of Figure 3-7, the main program code is statically bound to function `f()` so that if the function is recompiled, the main program does not see any change, as illustrated in Figure 3-8.
With dynamic binding, there is an indirection via the symbol table entry as shown in Figure 3-9. Thus any change in \( f() \) is reflected in the main program as shown in Figure 3-10. This indirect referencing may mean slower run-time performance. At run-time, the link traversal must check whether the destination is the function's symbol table entry or the code object. When the code object is reached, then the run-time system dynamically maps it in.
3.4.3. The Object Model

The object model needed to support orthogonally persistent C++ is little different from that of conventional C++. The one major difference is that every object in the persistent environment has a reference to its persistent type, which also acts as its run-time type identity. Additionally, because objects could have been faulted in from the persistent store, they may have an associated persistent identifier, PID which needs to be remembered.

The object format prescribed by the generic persistent store architecture is shown in Figure 3-11 [Brown & Morrison 92]. When committed to the persistent store, objects will conform to this format.

```
<table>
<thead>
<tr>
<th>lock word</th>
<th>number of pointers</th>
<th>total size (in words)</th>
<th>pointer fields</th>
<th>non-pointer fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>word -1</td>
<td>word 0</td>
<td>word 1</td>
<td>word 2 .. np+1</td>
<td>word np+2 .. sz-1</td>
</tr>
</tbody>
</table>
```

**Figure 3-11:** Generic store object format.

This format is simple yet flexible enough to be used for both data and code objects. From this base format, the storage layout we use for our C++ data objects is as in Figure 3-12.

```
<table>
<thead>
<tr>
<th>number of pointers</th>
<th>total size (in words)</th>
<th>type descriptor</th>
<th>pointers</th>
<th>non-pointers</th>
<th>bitmap</th>
<th>bitmap size</th>
<th>array size</th>
</tr>
</thead>
<tbody>
<tr>
<td>word 0</td>
<td>word 1</td>
<td>word 2 .. up+1</td>
<td>word np+2 .. sz-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Figure 3-12:** C++ store object format.

The type descriptor is a pointer that refers to the persistent type representation to which the C++ object belongs. Field $a_{sz}$ is the array size if the object is an array and $bm_{sz}$ is the size of a bitmap used to remember the object’s in-memory layout. Essentially, the bitmap is a sequence of bits, each of which is associated with a data member of the C++ object. If a data member is of pointer type, its associated bit value is 1 else the value is 0. The bitmap is needed to allow the store object to be shaped into a C++ object when faulted into memory. When faulted in, the store object is shaped into a memory object having a layout as shown in Figure 3-13. PID, $a_{sz}$, $data_{sz}$ and $bm_{sz}$ constitute a header present in every
Object. \textit{data\_sz} is the size of the C++ object in memory. Code objects have neither pointer nor bitmap parts.

\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{PID} & \textbf{type} & \textbf{a\_sz} & \textbf{data\_sz} & \textbf{bm\_sz} \\
\hline
\end{tabular}

\begin{itemize}
\item C-view
\item [object header]
\item actual C++ object
\end{itemize}

\textbf{Figure 3-13:} C++ memory object format.

\subsection*{3.4.4. Pointer Format and Semantics}

According to the C++ Reference Manual, de-referencing a pointer that does not point to a valid object results in undefined program behaviour. Therefore, to avoid pointer misinterpretation, pointer dereferences are subject to dynamic checking. This requires that our pointers carry enough information to allow such a check at the time of de-referencing. Therefore the pointer format used here is as shown in Figure 3-14.

\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{obj} & \textbf{index} & \textbf{lwb} & \textbf{upb} & \textbf{type} \\
\hline
\end{tabular}

\textbf{Figure 3-14:} Format of descriptor used as pointer.

The meaning of the fields are as follows: for any instance of such a descriptor, \textit{obj} is the memory address where the C++ memory object starts, \textit{index} is an offset used to point into the C++ object—for example in the case of arrays. Fields \textit{lwb} and \textit{upb} are the lower and upper bounds of the object and \textit{type} identifies the type that the descriptor is declared to point to. From \textit{type}, the size of the object can be obtained.

\section*{Pointer Arithmetic}

Given the above pointer format, pointer arithmetic can be checked at run-time and an exception thrown if the result of such arithmetic causes the execution to access memory that is not part of an object.

Similarly for arrays, since the name of an array is treated as a pointer to the zeroth element of the array, it is possible to exceed the bounds of a C++ array through careless arithmetic. In most traditional C++ implementations, a segmentation fault could occur, resulting in abnormal program termination. In a persistent environment this is unacceptable since the persistent store may be left in an inconsistent state. Worse still, if the abnormal behaviour goes undetected, persistent data could be unknowingly corrupted.

By implementing pointers as the descriptors described above, arrays will also undergo dynamic checks whenever their elements are accessed.

\section*{Pointer Conversion}

Through type conversion, either by type casting or using the functional notation, C++ allows data of a certain type to be interpreted as data of a different one. When type conversion is performed in a persistent store environment, store integrity becomes an issue. For example, pointers could be fabricated from integers and be used to access data in an uncontrolled fashion. Among other conversions, the following are allowed by the C++ Reference Manual:

1. A pointer may be explicitly converted to any integral type large enough to hold it. A value of an integral type may be explicitly converted to a pointer.
2. A pointer to a function may be explicitly converted to a pointer to an object provided the object pointer has enough bits to hold the function pointer. The reverse is also allowed. In either case, use of such a pointer may cause addressing exceptions, or worse if the pointer does not refer to suitable storage.

3. A pointer to a function may be converted to a pointer to a function of a different type. The effect of calling a function through a pointer to a function type that differs from the type used in the function definition, is undefined.

4. A pointer to one object type may be explicitly converted to a pointer to another object type. An addressing exception may result when that pointer is used.

To address the above, the following approach is adopted in this implementation:

- If a descriptor is converted into an integer, then the value of its \textit{obj} field becomes the integer value and the information held in \textit{index}, \textit{lwb}, \textit{upb} and \textit{type} are lost.
- If an integer is coerced into a pointer, the result is a dummy descriptor with all its fields initialised to zero or to the null pointer. Every subsequent dereference of this descriptor raises an exception. The same happens with a function pointer that is converted into a pointer to a different type.
- With respect to situation 4 described above, unless there is an inheritance relationship between the pointers involved, an exception is again raised.

\textbf{Pointing into Objects}

With the ability to point into objects, the issue arises as to what should happen if a pointer into an object becomes transitively reachable from a root of persistence? The pointer references valid data, and since persistence is orthogonal, the data member pointed to, should persist. Two possible approaches are:

1. Only that data member becomes persistent and is thereafter treated as an object in its own right.
2. The whole object containing the data member becomes persistent.

We chose to retain the entire object for the following reasons:

1. A later run may remember that the data member is part of an object and rightfully access the other members.
2. A pointer into an object is merely a computational path, it should not have the side-effect of dissecting an object to extract a particular field, i.e. it should not become an implicit object-slicing operator.
3. Within the same program execution, there may be another pointer accessing a different part of the object. If both pointers are within the closure of \textit{PS}, referential semantics should be preserved.

Also, in terms of implementation our pointers are descriptors with an index into the object. It is easier to retain the whole object. A similar situation arises with inheritance. If a base pointer references a derived object and becomes reachable from \textit{PS}, then the entire object should persist, not just the base part.

\textbf{3.4.5. Some Implementation Concerns}

This section documents implementation issues that arose during the prototype implementation that was done as part of this thesis work. The implementation part covers the following:

1. A persistent heap that sits on top of a generic persistent object store whose architecture follows the guidelines of [Brown & Morrison 92].
2. A command-line interface to interact with the environment, via which users may compile source code units, link object code and execute persistent applications.

3. Parts of a compiler that outputs code targeted for our orthogonally persistent environment. We modified an old EDG C++ front-end obtained under a non-disclosure agreement with the Edison Design Group. This aspect of the implementation is not complete.

Where is PS?

In the C++ model proposed, programmers are able to refer to the root of persistence by name, very much like in Napier88. However, the actual address of $PS$ is different for every invocation of the persistent heap, so that a static binding is undesirable. There is therefore the need to pass this information on to a program sometime before its execution. Napier88 provides programmers with a predefined function that can be called to obtain the value of $PS$ for the current store environment. In our C++ model, this can be achieved by having the declaration:

```
extern void * PS ;
```

as the first line in the source, either inserted by the compiler or coded in by the programmer. Having done so, since the persistent environment defines $PS$, an instance of the environment would have the actual address of $PS$ and at the time of mapping in a code object, the dynamic linker transparently resolves $PS$.

Stack Objects that Persist

For objects on the run-time stack that persist, a different code generation pattern is needed. For example, this object definition:

```
SomeClass an_object ( some_argument ) ;
```

is treated as though the following expression had been encountered:

```
SomeClass & an_object = * new SomeClass ( some_argument ) ;
```

This effectively converts the stack definition into an instantiation on the persistent heap. Incidentally, in doing so, the destructor will not be called when the variable name $an_{object}$ goes out of scope, which is a nice side-effect, since the object may persist. Therefore, the following differentiation needs to be made when objects from the stack persist:

1. Objects from the run-time stack, that are made to persist, are actually instantiated on the heap. The reason behind this decision is largely to ease our implementation efforts. Since the memory associated with objects on the run-time stack is relinquished whenever the current scope exits, it becomes difficult to retain those potentially persistent objects. The easiest way to support them in our implementation was to allocate them on the heap.

2. When the stack goes out of scope, the destructor is called on all the local objects except the ones that may persist. In support of this approach, it can be argued that since those objects have been retained, they should not be destroyed.

Persistent Functions and Code Objects

Functions are not allocated as heap objects. To support persistent functions, a shared object file, `.so` file, is generated for each compilation unit and placed in the persistent store as a code object. Thereafter, whenever a function in the code object is needed, the latter is unpacked via the file system and its `.so` file
is mapped in using the `dlopen` and `dlsym` dynamic linking routines. By default, persistent functions are statically linked to any executable that accesses them. Re-linking is needed to refer to the latest version of a code object. Old code objects are gradually removed as garbage collection discards those that are unreachable.

### 3.4.6. A Peep into Performance Overheads

The C/C++ programming community has traditionally been one that favours execution speed over safety. Thus in an implementation like the one proposed here, it is desirable that any change that might affect performance be investigated and quantified. The suggested changes where performance might be cause for concern are:

1. Dereferencing of pointers that now undergo run-time checks.
2. Array element access.
3. Union member access.
4. Calls to persistent functions.

To gain insight into the magnitudes of the overheads involved, an old EDG compiler was modified to output code that would run against a persistent heap. The EDG compiler is an old style compiler and outputs C as an intermediate code. The table below gives results of typical costs increases that code from the modified EDG compiler instance incurs over code from an unmodified one. All timing values are in seconds and the tests were carried out on a Sun IPX.

The figures for pointer, array, type-checks and tag updates were obtained by averaging over thirty test runs. The function call test calls a function to calculate and return a large prime number. The code object used for the persistent function was small enough to fit in one page size so that mapping involved bringing a single page into memory. It will not be surprising that performance suffers with increasing function code size, the degree of degradation being operating system dependent.

<table>
<thead>
<tr>
<th></th>
<th>no bounds checking</th>
<th>with bounds checking</th>
<th>cost increase %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000 pointer dereferences</td>
<td>0.512</td>
<td>0.560</td>
<td>9.4</td>
</tr>
<tr>
<td>1,000,000 array element accesses</td>
<td>0.548</td>
<td>0.592</td>
<td>8.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>no dynamic type check</th>
<th>with dynamic type check</th>
<th>cost increase %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessing</td>
<td>0.550</td>
<td>0.593</td>
<td>7.8</td>
</tr>
<tr>
<td>1,000,000 objects</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>No tag update</th>
<th>with tag update</th>
<th>cost increase %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000 union objects</td>
<td>0.210</td>
<td>0.240</td>
<td>14.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Statically linked</th>
<th>dynamic mapping</th>
<th>cost increase %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 function call*</td>
<td>0.025</td>
<td>0.028</td>
<td>12.0</td>
</tr>
</tbody>
</table>

* only calls to a function that needs to be mapped in from the store would incur this overhead.

### 3.5. Summary

This chapter has reviewed object-oriented concepts and terminology. The application of these concepts in the context of three object-oriented languages has been examined. These concepts are then used to show
that the Napier88 model of orthogonal persistence can be fitted to an OO language. C++ was used as an example of a "difficult" language and the bulk of the chapter presented the work done to support object-oriented orthogonally persistent languages, particularly C++, in single persistent object stores.

The rest of this thesis deals with similar computing environments but in a distributed setting. At the time of conducting the C++ experiment described in this chapter, Java has been gaining popularity and indeed, it appears that many persistence research projects are leaning towards Java. As will become obvious in chapter 6 onwards, this thesis follows the same trend.
Chapter 4

Distributed Persistence

4.1. Introduction

In the context of a software system, distribution usually implies the ability for an instance of that system to be spread over several processing elements or even address spaces. Such spatial distribution is adopted for three primary reasons that are not mutually exclusive. They are:

1. for the purpose of exploiting decentralised resources,
2. for modelling distributed processes and/or,
3. to harness the power of several computing units towards solving a single complex computation.

As previously noted, ideally, persistent object stores support the indefinite retention of programming objects in a computational environment. Over time however, the persistent object population tends to increase as more and more executions leave behind the result of their computations in the store. As explained in [Munro 93] the available address space of a single node may be insufficient and data may need to be distributed over a number of nodes.

Given that spatial distribution aims at spreading computational load or resources across a number of processing elements, therefore the solution to supporting the unbounded store concept has been and still is spatial distribution. Spatially distributed systems started from early attempts at making computers go faster, essentially by connecting processors into parallel arrangements. Flynn’s taxonomy [Flynn 66] is ubiquitous to most textbooks on distributed hardware. Ever since, distributed computing has evolved substantially—in particular, distribution as a software paradigm [Tanenbaum & Mullender 81, Birrell 85, Kessler & Livny 89, Li & Hudak 89, Stumm & Zhou 90, Bershad et al 93, Gropp et al 94, Keleher et al 94, Stevenson & Julin 94, Birrell et al 95, Feeley et al 95] is now as widespread, if not more so than hardware-based models. With distribution implemented in software, it is not surprising that the underlying hardware was not primarily designed to be an integral part of a distributed system. Apart from the well-known cost factors, this situation arose because it is increasingly the case that:

- Nowadays the typical organisation owns a number of generic workstations loosely connected by high-latency low-bandwidth networking. It is an attractive idea to harness their modest computing power yet, unlike the purpose-built multi-computer, any distributed capability other than their network card and inter-connect, must be supported in software.
• Organisations willing to make resources available from their site usually do so on an “as is” basis. Therefore, external users needing more than what the site offers may want to simultaneously exploit the resources from other sites to obtain complete service.

This chapter examines the properties of distributed software systems and, in the light of the knowledge outlined, advocates a model of wide-area distributed persistent systems.

4.2. Why Distribute Stores over Wide-Areas?

As noted earlier, all the workstations, servers or other machines with compute and network capacity, connected over geographical distances, constitute a significant source of compute potential. In other words, a wide-area network such as the Internet can be viewed as a global computer that could be readily exploited.

This global computer is an attractive platform that can be used to provide a distributed environment for low-cost computing across millions of computers that span the entire world. Two important properties of this computer are:

1. It appears to have no upper bound on the number of its nodes.
2. Its nodes are loosely connected by high latency and low bandwidth networking. This implies that this global computer is useful for coarse-grained interaction typified by small data transfer and relatively large computational activity.

This global computer is the platform of interest in this thesis. Having identified the target environment, there is now a need for a cooperation mechanism that will allow all the network nodes to interact and work in unison. The rest of this chapter investigates existing research efforts into distributed persistence models. Essentially it will be making an informed attempt to identify the model that best fits this environment. A primary criterion in this study of distributed persistence is the interaction mechanism supported and how resilient it is, given the unreliability of the network arrangement.

4.3. Properties of Distributed Systems

Distributed systems vary considerably in the properties they display. Depending on their underlying model and the motivations driving their designers, they make certain properties apparent to programmers whilst abstracting over others. Those properties that are of relevance to this thesis are outlined next.

4.3.1. Autonomy

Autonomy is the extent to which nodes must rely on one another for the entire system to work. Nodal autonomy varies substantially among distributed systems. Coupling can be as tight as the processors of a shared memory multi-processor e.g. the SGI Power Challenge [SGI 96], or as loose as a number of geographically distant servers in a wide-area arrangement. Different architectures achieve different problem-solving purposes and each one makes certain assumptions on link reliability and hence on the inter-dependence of its nodes.

Tight coupling typically features low latency, high bandwidth, total security and almost total link and processor reliability. Tightly coupled architectures are very good at supporting fine-grain distribution and parallelism. Unfortunately, it does not appear that such architectures will become more common. Undoubtedly some scientists constantly need highly specific supercomputers, multi-computers or their shared-memory counterparts. However, for the majority, these computational mastodons are largely
inappropriate for most applications, overly expensive and prone to obsolescence. What is becoming increasingly apparent on the other hand, is the "spread of the Internet" where homes and offices are getting connected to a global network through modest network cables and workstations. One can therefore envisage that such workstations connected to a public network will become as common as the telephone.

4.3.2. Location

The location of nodes in the distribution may either be visible or transparent and this in turn determines the mechanism of access at the programmer level. With visible node boundaries, the location of data and activity also becomes apparent to programmers and influences the programming paradigm used. To abstract over locality would mean that the system somehow hides the difference between local and remote access. In this thesis the terms locality and location are interchangeable.

4.3.3. Scalability and Dynamic Evolution

Dynamic evolution refers to the ability of a distributed system to add or withdraw processing nodes without bringing down the rest of the distribution. Scalability is the ability of a distributed system to adapt to a different distribution landscape. It is intimately related to dynamic evolution in that a system that is able to dynamically evolve is scalable. The converse does not always hold: scalable models do not necessarily have the ability to arbitrarily and dynamically evolve.

Tightly coupled architectures scale up to a certain point, beyond which a different mechanism for distributed interaction is needed. The extent of scalability depends on two factors. They are:

1. The distributed landscape changes in a way that can no longer be abstracted over by the implementation. For example, given a multi-computer that forms part of a departmental network, this would happen when crossing the address space boundary of the multi-computer to access another machine.
2. The distribution is subject to restrictions that are beyond the control of the implementor of the distributed system. This typically happens when the distributed system potentially spans multiple organisations and each organisation does not wish to allow unconditional access to its resources. Organisational boundaries fall outside the realm of Computer Science and it is not possible to abstract over them in any distributed system. They are to be reckoned with. Many examples of such systems are found in the Web-based application domain.

4.3.4. Fault Tolerance

The degree of fault-tolerance is the extent to which a distributed system may continue operation despite nodal or link failure. Fault-tolerance is usually inversely related to coupling so that the tighter the coupling among processing elements, the more likely will the failure of one, affect the others. Possibly therefore, in a landscape where the nodes are autonomous, the failure of one node could be isolated well enough so as to shield the processing ability of other nodes.

4.4. Models of Distributed Persistence

From around the time when PS-algol became established as a stable persistent programming language system, investigations were directed towards a distributed version of PS-algol [Wai 89]. Since then, much research effort has been spent on exploring and implementing some form of distributed persistent storage.
Many different architectures to support the distribution of persistent stores have resulted from the effort of research groups world-wide. Broadly, they fall under one of three models of distributed persistent object stores as identified in [Munro 93]. They are,

- the one-world model, so called because it abstracts over object location and ideally allows the programmer to reference remote objects with the same degree of programming ease as with local ones,
- the federated model, where store boundaries are visible but where inter-store references are allowed. Over time, stores become more tightly coupled as more and more references cross boundaries and create dependencies, and,
- the confederated model, which is essentially a federated arrangement that preserves store autonomy by preventing references from crossing stores. Store boundaries are visible and direct inter-store references are prohibited, usually by the absence of operations that accept or return references.

This chapter examines instances of each of these models. Although the above categories provide a neat classification, some systems neither fit into the federated nor the confederated class. This is because although they disallow direct inter-store references, they feature other characteristics that bind stores into associations thereby compromising autonomy. In this chapter they are termed the "hybrids". Two of them, namely PJRMI [Sun 99a] and PerDiS [Kloosterman & Shapiro 99], are of noteworthy consideration.

4.4.1. The One-world Model

Central to the persistence paradigm, is the transparency of locality. Persistence abstracts over the storage and location properties of objects [Brown 88], so that programmers need not be aware of either. The one-world model endeavours to extend this abstraction to encompass any space, especially distributed ones. The primary implication is that ideally, an inspection of the syntax and semantics of operations on remote objects does not distinguish them from local objects. In fact, to the programmer, the terms “remote” and “local” become meaningless with respect to objects in a one-world persistent space. It follows therefore that in this model, distributed programming over a number of stores degenerates into conventional programming in a single persistent store.

In a failure-free network, the one-world model is attractive because of its simplicity and elegance, and also because it espouses well the ideals of orthogonal persistence. Not only is the programmer relieved from working out how to store and retrieve objects, but the system also makes all objects available upon reference no matter how geographically distant they happen to be.

However, the reality is that an implementation of the model has many difficulties to reckon with. Some of the major ones are:

- The provision of a single object space inevitably encounters scalability problems, especially if the implementation eventually evolves over a wide-area network of substantial geographical span. An example of such growth is the World-Wide Web that now encompasses planet Earth. A one-world implementation would therefore scale up to a certain point, say the organisational boundary, beyond which a different mechanism is needed to cope with the different distribution landscape.
- Location transparency means that programmers rely on some form of unique name or identifier to refer to objects. Such a uniform identification scheme is not easy to implement considering that persistent stores are typically large long-lived environments.
4. Distributed Persistence

- Even in a single store, garbage collection becomes increasingly expensive with increasing store population. A distributed arrangement may exacerbate the problem.
- Each node relinquishes its security and autonomy. For some application domains, this is undesirable. For example, a product manufacturer may want to encourage its customers to place orders on-line without having to open up its information processing space to the world.
- Stability and domino-effect roll-backs as already explained in [Munro 93]. A one-world space should convey the illusion of all-or-nothing stability, in other words, when a commit happens, either the entire space commits successfully or it completely rolls back. With an implementation that grows to be as big as the World-Wide Web, stability could require the whole planet to commit or roll back.

And most important of all, failure—whether of links or of nodes, can be impossible to mask. Failure, or even mere network delay can introduce additional problems in the form of,

- Network partitioning.
- Time-warps.

Thus for an implementation of this model to succeed, there is a need to completely abstract over failure. Either the physical network is made totally reliable or the software networking layers are somehow equipped to take appropriate action in response to every type of failure. Both approaches are currently impractical in wide-area networks of commodity workstations and servers.

The next three subsections examine existing one-world implementations.

**The Multi-Computer Texas Experiment**

The MC-Texas project [Blackburn & Stanton 96] was an experiment to develop a multi-computer version of the Texas persistent store examined earlier. MC-Texas investigated the possibility of providing a single store image of a distributed store. Each node would see the same logically single but physically distributed store. This distributed store would also have the ability to scale when the underlying hardware is expanded with extra processors and other system resources. It was posited that store scalability would then only be limited by the scalability of the physical network.

The design is targeted for multi-computers where many tightly-coupled processors, each with its own memory and possibly its own disk, constitute the computing environment. For simplicity, it adopts a distributed single process model—only one process actually exists, and a copy of it executes on each node of the machine. Each node manages all pages that are held locally and the nodes communicate via message passing using MPI [Gropp et al 94].

In order to support persistent distributed shared memory, MC-Texas implements a simple mechanism based on page locks. For each memory-resident page in the persistent space, there is one immutable original and zero or more mutable copies. When a node locks an object, the node obtains the master page hosting the object and retains it until it is required by another node. This potentially leads to false sharing whereby all the objects on a page are locked whenever a page is locked, even though only one object is needed by the locking node. Experiments by Blackburn and Stanton showed that very high levels of false sharing occur because the page-based granularity is excessively coarse for concurrency based on objects at user-code level. In other words, some aspects of the architecture make it unsuitable for scalability.
The distributed persistent heap in MC-Texas necessitates both distributed free-list and distributed persistent-to-virtual address map management. Independent free-lists mean that a node may run out of persistent heap space although space is still available on other nodes.

As chapter 2 examined, the Texas store traps attempts to refer to a page that is not in virtual memory by ensuring that for every persistent address, the corresponding page is access protected. Writing to pages is detected by initially giving pages a read-only status. When a write occurs, the page is flagged dirty until the next checkpoint, at which time it is given read-only status again. In MC-Texas, the use of master and copy pages necessitates a central map server to perform concurrency control.

MC Texas was implemented on a Fujitsu AP1000 multi-computer and whilst largely successful, one conclusion drawn by the implementors, was that the architecture was not scalable. Strictly from a feasibility viewpoint, it can scale up to include extra nodes added to the machine but is essentially restricted to the physical bounds of the Fujitsu AP1000.

An implementation based on a network of workstations was also carried out yet one would expect that scalability beyond the local area out to a wide area network, would be problematic. Some aspects of the architecture are still centralised, such as the map server, whilst others such as address swizzling, may require fine-grained collaboration by several nodes at the same time.

\textit{CASPER}\\
Whilst MC-Texas allowed a single user to harness the power of a multicomputer through a store distributed over the multi-computer, research has also been done on supporting multiple user accesses to a central store. The Cached Architecture Supporting Persistence, CASPER, was such a project [Vaughan \textit{et al} 92]. CASPER developed a client-server architecture to support Napier88. In this system, a number of clients run concurrently against a single server that hosts the central persistent store. Each client consists of an instance of the Napier88 interpreter running an application, a local page cache holding copies of data pages from the central store and some supporting routines for page migration to and from the server. The store itself is a server process with three active components: a handler of requests from clients, a manager of store pages and a garbage collector. The stable medium is also part of the server.

In a client-server architecture such as CASPER, concurrency control and coherence are major issues. CASPER solves the concurrency issue by making store pages solely accessible by the server. It is responsible for supplying coherent pages to clients on demand, and it also maintains store integrity. A stabilisation protocol evolves the store along some path of consecutive consistent states. CASPER attempts to find "partial cuts" through the persistent space and stabilise only the pages that are associated with or affected by the client making the stabilise. To achieve this, the server maintains lists of interdependent clients that must be stabilised together. A set of mutually dependent clients is termed an \textit{association} and whenever an associate stabilises, the entire association participates.

Whilst CASPER is a distributed system, its main achievement has been as follows:

- It shows the feasibility of association-based stabilisation where it is often possible to only stabilise part of the persistent space. This is in marked contrast to both the "stop-the-world" approach and the forced checkpoint approach where a client is forced to commit whenever one of its modified pages is requested by another node,
- It breaks the single user restriction of the traditional Napier88 computing environment and allows multiple clients concurrent access to a store.
However, the CASPER store is centralised. In large implementations, the higher the number of clients, the busier the server. One may therefore expect that performance will degrade with increasing numbers of clients. The Thor project examined below attempts to take the model beyond a single persistent store by distributing the persistent object space over a number of servers, each of which supports several clients.

**Thor**

The Thor Persistent Object Store [Liskov et al 93] is an impressive endeavour to develop a seemingly single yet distributed large universe of persistent objects. This account focuses on those features that are of relevance to a wide-area distribution. The following features characterise the Thor design:

- Thor runs in an environment of nodes connected by a network. The network is likely to be a number of local area networks connected by wide area technology.
- Some nodes are servers where persistent objects are hosted whilst others are clients, where users run their programs. A node can be both a server and a client but this is not expected to be common and the design is tuned towards distinct server and client nodes. A client node would usually be a user's workstation.
- Front-end interfaces run at client nodes, and back-ends and object-repositories run on server nodes. Users only interact with Thor via the front-end. A front-end accepts requests for Thor objects from user programs and in turn requests a back-end for some persistent object residing in a repository.
- An OO language, Theta, is the language used to define and implement Thor object types and methods. There has also been an attempt to use Java.

The design targets geographically distributed servers, each of which services a number of clients. Clients are expected to rely primarily on nearby servers. The judicious choice of a server by clients is important and causes the architecture to benefit from locality of interaction. Thor could be seen as being a collection of CASPER servers over which locality has again been abstracted. In other words, Thor extends the CASPER model to allow a one-world distribution over wide inter-connects of LANs.

Thor adopts a safe approach when it comes to interaction with objects—objects are encapsulated so that clients can only access them by calling their methods. In so doing, Thor achieves safe sharing both in time and across space. Furthermore, objects can only be accessed inside atomic transactions. A transaction consists of a number of calls to methods of Thor objects and ends with a commit or an abort.

Although a user manual was produced, a WAN-based implementation has yet to be released. Among the many issues that are relevant to large one-world implementations, it is still unclear how Thor addresses the following:

- Management of the global space—in particular, the practicality of commit operations that span several distant servers.
- Authorisation and security when users access objects.
- Behaviour of replicated objects in the presence of network partitioning.

At the time of writing up, much of the techniques that Thor proposes to use are still under development. Preliminary performance results indicate that a convincing full-blown implementation could eventuate one day. Recent papers published on Thor indicate that:

1. Fine grained data, typically of the size of objects normally handled by common OO languages, need not lead to unacceptable overheads in a wide-area distribution.
2. It may be feasible to rely on dynamic type-checking every time a persistent object is accessed.

4.4.2. The Confederated Model

In the federated model, there is no attempt at uniting the stores into a uniform world. Computation is by default local to a store, store boundaries are made apparent to the programmer, and two stores interact only if they are both willing to cooperate, not because they are part of the confederacy. Unlike the one-world model, accessing a remote store is an explicit programming activity that is subject to permission. This makes the programmer aware of what resources are remote, hence are more expensive to access, and are potentially unreachable because some network failure may occur.

Once a remote store is accessed, the operations on available resources hosted there are likely to be different as well. Usually, only a restricted set of operations is allowed. Such restrictions are aimed at preserving the autonomy of the remote store. In general, operations at the remote store do not return any local reference to the store that requested service.

This model is attractive because it imposes few rules on the global store community. In particular, it mainly requires stores to be operational and to be accessible via some common communication medium. It does not require stores to be always operational neither does it require the medium to be reliable. In other words, the model accommodates the reality of failure.

**STACOS**

The St Andrews Confederated Object Store, STACOS [Munro 93], exemplifies the federated model. STACOS was developed by Munro to enable autonomous Napier 88 systems to interact in ways that do not allow stores to become inter-dependent [Munro 93]. Its base model is slightly restrictive in that services are limited to interrogating remote stores and to the copying of object closures from one store to another. The strength of STACOS is that each constituent store in the confederacy is no different from a stand-alone Napier 88 system, except for its willingness to cooperate. The cooperation expected from a store involves the following:

- listening to the network,
- being able to browse other stores,
- allowing other stores to browse its population,
- obtain closures from remote stores and,
- give copies of local closures to remote stores on request.

Clearly, none of the above is likely to create associations between stores and stores can always stabilise or garbage collect independently. Also, it is to be noted that the last two operations are type-secure.

Each STACOS node maintains a database of symbolic names of remote stores and associated database management routines, and is equipped with a number of procedures and data structures to support distributed interaction. To enable a Napier 88 client to connect with a remote store, the local store provides a procedure, `openRemoteStore`, that can be called with a symbolic name as parameter. If the connection request succeeds, the procedure returns three procedures: `scan`, `remoteCopy` and `closeRemotePack`, that can thereafter be used to browse the remote store, copy remote objects and to eventually close the connection.

Except for `remoteClosePack`, all these procedures may fail, in which case the return value is a string that the programmer can inspect to understand the semantics of the error.
4. Distributed Persistence

A possible weakness of STACOS lies in the way a store freely opens up its object population after accepting a connection. Once procedure openRemoteStore is successfully called by a client, scan and remoteCopy empower the client to inspect and copy the entire remote object population. It is unclear whether the remote store has any control over the visibility and accessibility of its objects. If not, STACOS nodes may be vulnerable in a hostile networking environment.

Additionally, remoteCopy is an operation defined on stores. It follows that an object in a remote store has no say in whether it should be copied across the network or not.

4.4.3. The Federated Model

This model of distributed store attempts to compromise between the above two extremes. It aims to retain the attractive simplicity of programming in the one-world model whilst recognising that allowing the location of an object to be determined can lead to some degree of programmer-induced efficiency. A simple example is the ability to discover print queue locations before a printing program schedules a document for printing.

The primary difference between the confederated and the federated model is that the latter allows references to cross store boundaries. In the STACOS project, the need to reduce the amount of data transferred in deep-copy operations was acknowledged. Federated implementations avoid this by passing references to remote data. This results in stores becoming inter-dependent and thereafter, they must be managed together. DPS-algol is an instance of the federated model and is examined next.

**DPS-algol**

The Distributed PS-algol project [Wai 89] investigated the distribution of persistent spaces over several cooperating nodes. DPS-algol supported the notion of a single address space but also provided programmers with the ability to uncover the locality of data. The motivation for the DPS-algol approach was that whilst abstraction over locality made distributed programming as easy as conventional programming, the visibility of node boundaries would allow programmers to develop more efficient applications. It was expected that knowledge of where resources are located, would influence programmers to code their applications so as to take advantage of such knowledge. DPS-algol therefore features the concept of a *locality* as a construct in the language. Operations are available to create new localities as well as to discover the locality of an object. For example,

```
let my_resources = newlocality
```

creates a new locality called *my_resources*, and

```
let there = locality some_object
```

can be used to discover an existing locality, the one in which *some_object* resides, whilst,

```
add there to my_resources
```

causes *there*, which is a possibly remote locality, to be reachable from *my_resources*. Two predefined localities are provided: *universe*, the root of all localities and *here*, the locality of the executing process.

In DPS-algol, two operations, namely *transcopy* and *assign* allow data to be copied across localities. [Wai 89] gives more details of these operations, suffice to say that these operations may result in objects at a node containing direct pointers to sub-objects that reside at remote nodes. This is because among the rules applied during copying, are the following four:
1. If a locality is being copied, only a pointer to the locality is copied.
2. For vectors, only the top level is copied to the remote locality. So a vector of integers would be copied in its entirety whereas for a vector of pointer to integers, only the top level is copied causing the copied pointers to still refer to the original integers across the localities.
3. A procedure is copied in such a way that the procedure's original environment is shared, again across locality boundaries.
4. Structures are copied using the same rule as for vectors.

As a result of applying any of the above rules, a pointer in one locality may reference an object hosted in another locality. This in itself is not a problem if the two localities are on the same physical machine. Unfortunately, this physical closeness cannot be guaranteed and inter-locality references may well cross physical boundaries. For example,

\texttt{transcopy my\_data to remote\_locality}

copies at least the top-level object referenced by \texttt{my\_data} to locality \texttt{remote\_locality}. Whether the entire closure is copied or only a top-level copy is made, depends on the type of \texttt{my\_data}. Values of primitive type and rectangular matrices of pixel, representing images, are entirely copied whilst for the more complex or structured types only the top level is copied. This \texttt{transcopy} activity is the cause of "pointer leakage" and of loss of autonomy among stores. Thus despite being individual, some stores may need to stabilise together and should any one store roll back, all other stores that share closures with it would be affected. The absence of synchronisation in the model makes it hard to manage large implementations because there is a constant need to track down dependencies and find out which stores need to stabilise together.

Notwithstanding the above, the noteworthy features of DPS-algo are:
1. the introduction of the locality concept into persistent environments,
2. the realistic recognition that remote operations may fail. To that effect, \texttt{transcopy} and \texttt{assign} raise an exception allowing the programmer to implement corrective measures,
3. the avoidance of phantom copying and,
4. preservation of referential integrity among copies of object graphs unless deep-copying is explicitly coded in by the programmer,

all of which are of value and ought to be retained in future distributed persistent system development.

4.4.4. The Hybrids

Some distributed systems, examined as part of the work described in this thesis, were found to display mixed properties. They are the hybrids. Three of them are PerDiS, Argus and PJRMI, examined next. Both Argus and PJRMI do not allow direct references to cross address spaces and are in that sense, confederated. However, in Argus, when a program executes, every node where the program causes state changes becomes involved in a two-phase commit that creates nodal dependencies. In PJRMI, clients and remote object implementations interact through pairs of static remote interfaces. The need to manage these pairs of interfaces again causes interacting stores to lose their autonomy. PJRMI provides \texttt{remote references} for programmers to refer to remote objects. Although they behave like normal references an important difference is that the use of a remote references can cause exceptions to be thrown, allowing the application to recover. However, the existence of a remote reference in one client store is a guarantee that
a referent exists in some remote server store. For as long as the remote reference persists, the server is required to retain the referent.

PerDis is a one-world model that localises its object space into clusters. Clusters are visible to programmers and must be explicitly opened before their root objects can be accessed. Once a root object is retrieved, accessing the rest of the cluster population from it is by ordinary pointer de-referencing. Clusters are identified by way of Uniform Resource Locators, URLs [CERN 90].

**Argus**

In existing literature, Argus [Liskov et al 87] is classified as a federated model. Argus programs are centred around guardians and actions. The Argus guardian is an abstraction for a locality that encapsulates data and processes. Argus data is not shared among guardians, but every data item wholly resides in one and only one guardian. Thus unlike the DPS-algol locality, a guardian encapsulates its data and defines handlers, which are the service access points via which other guardians interact. To interact with a guardian, other guardians call its handlers which are then executed and access the encapsulated resources. Handlers are the only mechanism whereby a guardian communicates with other guardians and operates on data hosted by them.

Given a number of guardians each offering some resources, a program may now interact with them by remotely calling handlers defined at those guardians. An action is the Argus abstraction of a program and it has an atomic, serialisable, recoverable and total effect. Here, total means all or nothing. As the action executes, it potentially modifies data through remote calls to handlers of several guardians. If the action succeeds, then a two-phase commit ensures that all the guardians affected by the action commit their changes to permanent storage or else they all roll-back together.

One good feature of Argus is that guardians prevent pointer leaks, enabling the guardians to be self-contained and easily managed. This feature has been adopted into the confederated model and remains one of the strengths of the model. Prevention is achieved through encapsulation of guardian data and by passing all handler arguments by copy. This unfortunately translates into a loss of referential integrity whenever guardians have to communicate data between one another—when a reference to an object is passed to a handler call, a copy of the top-level object together with its entire closure is made and transmitted to the remote guardian. Accordingly, computation where handlers from different guardians need to operate on a single data item, may become difficult to model.

Additionally, guardians are intended to be units of tightly coupled data and processes. Under this assumption it is acceptable that an action binds several guardians into an association that is close enough to require all to commit or roll-back together. However, in wide area arrangements with frequent network hiccups, the implications of retaining this assumption become uncertain.

A guardian is also a static entity in that, once deployed, the resources it represents and the handlers it defines cannot be changed. In other words guardians cannot dynamically reconfigure themselves.

**PJRMI**

PJrmi is a port of Sun Microsystems' Java RMI [Sun 97] to a persistent environment, namely PJama [Sun 99a]. The Java RMI model itself borrows extensively from Modula-3's Network Object [Birrell et al 95]. In the PJRMI model, a PJama store that has some remote objects to export runs any program that maintains a long-lived session of the virtual machine. For example one such program can
simply call `java.lang.Thread.sleep` repeatedly. For as long as that machine instance lasts, the remotely
invokable objects are automatically available for use. Then, clients from other stores can do a look-up on
a remote object and invoke methods defined on it. The entire scheme operates as follows. For every object
whose methods may be remotely invokable:

- There is at least one remote interface that declares the remotely invokable methods, and exactly one
  implementation class. Client applications always interact with interfaces, never directly with the
  actual object.
- For each implementation class, there is a stub/skeleton pair of classes through which client and server
  communicate.

A store that has such an object to offer to other stores is equipped with a registry, and to it an instance of
the implementation class is persistently bound. This object is the one whose methods are remotely
invokable. Binding is through the usual RMI mechanism, namely `java.rmi.Naming.rebind`. After being so
equipped, the store can now become a ready server by executing any application that maintains a long-
lived session of the virtual machine. Thereafter, applications in potentially remote stores can look up the
object and invoke those methods declared in the interface located at their site.

Local objects used as parameters to remote invocations are passed by copy. That is, the contents
of a local object parameter, is copied before making the call to the remote object. Similarly, return values
of remote invocations are transmitted back by copy. In the latter case, the contents of the result are
transmitted back and a new object is created in the calling store.

In PJRMI, client and server stores interact through a stub/skeleton layer. Each remote object has
an associated pair of classes: on the client side it is called a stub class and on the server side, it is a
skeleton class. The stub class is used for arguments marshalling and for asking lower PJRMI layers to call
the server side. On the server side, the skeleton class is used to unpack the arguments before making the
up-call to the remote object implementation. Upon call completion, the skeleton class is then used for
marshalling the result which is to be sent to the waiting client. Once the result is received by the client
store, the stub class is finally used to unmarshal the result. The result can be an exception, in which case
the stub class will throw the exception in the client.

Distributed programming with PJRMI does not cause direct references to cross store boundaries
and in that respect the model is confederated. However, PJRMI’s reliance on compiler-generated stub and
skeleton classes for every remotely invokable object creates undesirable associations between the client
and the server stores. Additionally, every reference that persists in a client causes the remote object to
persist as well. This again causes loss of autonomy at the server store.

In some cases, the need to always interact with a remote object via a pair of interfaces prohibits
on-the-fly creation of remotely invokable objects for callback situations. In particular, the restriction
applies to objects that were not originally designed to implement a remote interface. This is because the
creation of interfaces has recourse to activities that are outside the language domain. Even if interfaces
could be dynamically created from within the language, currently, `java.rmi.Naming.rebind` requires the
object to be named. This causes every bound object to become globally visible and may be undesirable,
although it does not appear to be difficult to have an overloaded `rebind` that does not need a `String` name
argument and returns a `Remote` reference to its caller.

The client stub class is generated from the implementation class, which is usually hosted at the
server store. Somehow, that stub must be distributed to each client before they can invoke the remote
methods. This can be done manually at the time of deploying the distributed application or automatically
at run-time if the java.rmi.server.codebase property has been appropriately set. That stub eventually becomes part of the persistent environment and in the event modifications are made to the remote object implementation, the existence of the old stub in the persistent environment of the client makes it difficult for the client to access the new version of the stub.

**PerDiS—Persistent Distributed Store**

PerDiS [Kloosterman & Shapiro 99] is an endeavour to provide large-scale transparent distributed persistence targeted at the virtual enterprise—a virtual enterprise being a consortium of organisations working together on some project. The organisations are separate entities and are possibly based in different countries. We noted earlier that the scalability of the MC-Texas Store is limited to the physical bounds of the AP1000 multi-computer for which the store was built. PerDiS aims at providing a similar single store image but one that operates over wide-areas. Also, whilst MC-Texas adopted the distributed single process approach, PerDiS allows multiple, possibly competing processes, to execute against the distributed persistent store.

PerDiS organises its distributed persistent space into clusters. Clusters are visible at the programming level and act as the unit of naming and storage. URL names are used to identify clusters, e.g. pds://alpha.inria.fr/aCluster, and these must be explicitly opened. Thereafter, their contents can be manipulated, typically by first retrieving a named root object. Once this is done, existing objects can be accessed through the root, or new ones allocated inside the opened clusters.

Processes access objects by “holding” on to pages thereby indicating to others that they need the said pages. Clusters are implemented as operating system files, they are made up of segments of pages, and these in turn contain objects.

Similar to MC-Texas, PerDiS’s distributed shared storage approach ensures that programs accessing the store are provided with direct in-memory access to potentially remote objects. Again, this is achieved through page-fault detection and virtual memory mapping techniques: if a pointer to a remote object is traversed, the latter is automatically brought into the program’s memory.

The latest annual reports from the PerDiS research group suggest that implementation has been successful. Indeed, existing applications e.g. xfig have been ported to it and, test results indicate that performance would largely be acceptable for similar applications used in collaborative CAD environments. However, the tests were conducted on a local area network. It is unclear what kind of results would be obtained if the experiment is translated to wide-area interactions. In addition, being a one-world model implemented atop physically distributed memory, PerDiS is likely to suffer from the same bottlenecks as MC-Texas. In particular, we have in mind cluster contention at the shared memory level and false sharing in the application process spaces. Performance of global checkpoints is also unexplored at this stage. And finally, given that PerDiS aims at easing distributed programming in the virtual enterprise, it is unclear how the existence of organisational barriers are acknowledged and respected in the PerDiS model.

4.5. **Which Model?**

Clearly, the one-world model offers the best programming environment, both in terms of simplicity of programming and in its ability to abstract over the physical properties of objects—also a longstanding strength of orthogonal persistence. To date, there are however several obstacles that still impede the development of one-world implementations on a large-scale basis. Four of them are:
1. Scalability. Scalability has always been an issue in distributed systems despite the fact that increasingly reliable and fast networking technology is continually being developed.

2. Efficiency. If all distribution is abstracted over, then it becomes difficult to write efficient distributed programs.

3. Organisational barriers. Unfortunately, these are an issue that fall outside the realm of computer science and are not dealt with in most distribution models. They should be reckoned with, instead of being ignored.

4. Speed of light limitation. This limitation convinces us that despite the advances in WAN technology noted in 1 above, latency can never be fixed.

In view of the current state of network technology, especially with respect to wide-area implementations, this thesis supports the view that the confederated model is most appropriate for supporting distribution among cooperating autonomous persistent object stores. From an efficiency point of view, transparency as advocated in the one-world model is virtually impossible to engineer. Additionally, total abstraction over failure is unrealistic. Therefore, it is not viable to abstract over locality because such inefficiency frustrates users. Henceforth the remaining chapters will concentrate on discussing a confederated arrangement of interacting stores.

Although the adoption of the confederated model might feel like a step backwards from the one-world holy grail, we feel that ignoring organisational barriers merely for the sake of convenience, is unacceptable. We also note that nothing in Munro’s model prevents a store in the confederacy from being a single store image in the style of MC-Texas. In other words, the confederated model is able to incorporate one-world implementations as individual nodes in the distribution.

Additionally the discussion that led to choosing the confederated model examined several existing systems. These systems all had strengths and weaknesses and their strengths must be retained whilst their weaknesses avoided. In particular, these properties should be retained:

1. The autonomy of nodes as exemplified by STACOS nodes and Argus guardians.
2. The ability to remotely invoke the methods of an object as in PJRMI.
3. A mechanism to retain referential integrity as in CASPER.
4. An awareness of resource localisation as in DPS-algol and Argus.
5. Encapsulation as in Argus guardians.
6. Identifiable failure conditions that allow the programmer to opt for different computational paths.

And we seek to avoid:

1. Static stub/skeleton class pairs for every remote object type.
2. Pointers that cross address spaces. Yet, preventing such pointers by only supporting by-copy parameter-passing is overly restrictive. There is a need to support some form of by-reference semantics in remote invocations without resorting to direct pointers.
3. Uncontrolled access to a store’s objects once it grants connection to a remote client.
4. Static configuration of localities. In other words we seek to allow localities to be reconfigured even after they have been deployed for operation.

The federated model was deemed unacceptable because of the existing knowledge on the difficulty of managing DPS-algol stores. This difficulty stems from the cross-locality references that make store
management and garbage collection more complex than in confederated stores. Garbage collection in one-world models is also complex and may be expensive if the scale is significant.

4.6. Parameter Passing Conventions

One important feature of the three models of distributed persistence discussed above is the semantics of passing parameters in remote calls. Upon such semantics rest the preservation of nodal autonomy. Parameter passing follows one of two main conventions, namely by-copy and by-reference. The first scheme is based on value-transmission whilst in the second, a reference to the actual parameter is provided by the caller.

4.6.1. Call by Copy

In the call-by-copy convention, formal parameters act as local variables or constants in the callee. Depending on how the formal parameter values affect the actual parameters, if at all, when the call returns, call-by-copy can be further categorised into by-value, by-result and by-value-result.

With call-by-value, formal parameters are initialised with the value of the actual arguments to which they correspond. No information flows back to the caller via the actual arguments when the call returns. Call-by-result operates the other way round and information flows via parameters from the called routine to the caller. Formal parameters are not initialised at call time but when the call returns, the contents of formal parameters are copied into the actual arguments. Call-by-value-result combines the semantics of both. As an example, the Ada in, out and in out modes may be implemented under these three by-copy classifications.

In a distributed environment, call-by-copy ensures that pointers do not cross address spaces. In the previous study of existing systems, STACOS and Argus employ call-by-copy. PJRMI does so when transmitting local objects. Unfortunately some computational models require several computations to access shared remote data. Making by-copy calls does not support the requirement of such models.

4.6.2. Call by Reference

In call-by-reference the caller passes the memory address of the actual parameter to the called routine or callee. In the callee then, a reference to the formal parameter will access the memory location whose address was passed. In other words, the formal parameter is the same as the actual parameter. The memory address itself is passed by value.

Thus during the activation of the callee, the actual parameter is shared with the caller and is directly accessible to both. This is of consequence in distributed persistent environments for the following reason: if the called routine causes the reference to persist locally, then for as long as the reference persists, that store is dependent on the store from which the reference originated. Once such a reference persists, the store from which it originated cannot discard the referenced object. Such persistent references destroy store autonomy.

However, despite the creation of dependencies among stores, parameter-passing by reference is advantageous in that,

1. the mechanism is generally efficient because a minimal amount of data is transmitted,
2. true sharing is enabled—both the caller and the called routine have a reference to the same data, and
3. sometimes call by reference is unavoidable.
In CASPER and MC-Texas, by-reference semantics would apply when a reference or a pointer to an object is accessed from different nodes.

4.6.3. Call by Substitution

Migration-by-substitution [Mira da Silva 96] is a novel mechanism that emerged from the FIDE endeavour [FIDE 90]. Usual distributed systems terminology applies the term “migration” to data or processes, to convey the semantics of data or process movement from one address space to another. In other words the entity leaves its home node and establishes itself somewhere else in the distribution. In migration-by-substitution, no such migration occurs and to be more accurate, the term call-by-substitution will be used henceforth.

STACOS revealed that some object closures used in remote calls were large and made call-by-copy transfers voluminous. To that effect, call-by-substitution was developed. In this scheme, two stores are involved, the store where the caller executes and the remote store that hosts the called routine. Prior to the call, the two stores have an agreement on a logical name that identifies an object that will be “transmitted” by the caller. This same name identifies a similar object in the remote store. At the time of the call, the caller then transmits that name as a surrogate. At the receiving end, the target store uses the name to identify the local object mentioned in the agreement and then calls the routine with the local object as a parameter.

Each store in the distribution maintains a database of name-object tuples that can be used as substitutes for remote calls. The database is defined by application programmers who use the store. Though it appears that the scheme is primarily intended for pairs of stores, it can work in multi-store interactions provided all the stores agree on the names together. Unexpected behaviour is likely to be observed otherwise. For example two stores A and B agree on a name X for some object, whilst separately stores C and D agree on it for some other object of similar type but different semantics. If A now calls a routine in C and provides an actual parameter from which X is reachable, then when C receives name X, it may associate its local object with the name. Because both Xs are of the similar structure, the call will type-check correctly. From existing literature, it is unclear whether the call-by-substitution mechanism incorporates a means to remember and manage these store-pairs. Among the weaknesses of call-by-substitution, are the following four:

1. All stores involved in making and servicing remote calls are bound by agreement on name-object tuples. Very often, the name of an object, especially if represented by a string, does not convey enough information about the semantics of that object.
2. By-reference semantics that allow true sharing is still not achieved.
3. It is unclear if names have to be unique across the entire distribution.
4. Programmers need access to both source and target stores.

The application of call-by-substitution would be restricted to those objects that are common denominators to all stores. This is likely to be predefined system objects e.g. those that are part of the standard libraries.

4.6.4. Call by Indirection

Although call-by-substitution has been proposed as a compromise between by-copy and by-reference semantics, it probably best serves the purpose of avoiding transfers of large closures in by-copy calls. It is efficient and should be used in situations where the illusion of sharing is adequate. There is still a need for a resilient parameter-passing convention that supports true sharing whilst preserving store autonomy.
Additionally, there are some circumstances where call-by-value should be avoided. The classic example is that of a file or socket object with an embedded descriptor. This descriptor identifies a resource that is external to the language environment—usually one that is administered by the local operating system. It would not make sense to transmit such an object by value.

This thesis advocates that call-by-indirection could be such a convention. Detailed treatment of the indirection concept follows in the next chapter. In this convention, an abstraction that acts as a reference for an object is passed in a remote request. Because the abstraction refers to the actual object over which it abstracts, not just a local substitute, true sharing is achieved. Yet because the abstraction does not use direct pointers, autonomy is preserved. Furthermore, to cater for resilience, a number of exceptions are defined and these may be thrown whenever the abstraction operates under unreliable networking conditions.

### 4.7. Paradigm Shift

Notwithstanding the importance of the properties of a distributed persistence model, ultimately an instance of any model is only useful if programmers are comfortable with it. Typically, each instance provides some mechanism to access and operate on remote data. Herein lies the strength of the one-world model. Ideally, the mechanism of access is the same for both local and remote data so that the programmer need not care about the locality of data. On the other hand, federated and confederated implementations offer a set of operations that are specific to remote data. These operations are likely to be different so that there is a shift in programming paradigm whenever the programmer has to use them instead of the “normal” operations applicable to local data. The following table illustrates the relationship among all three models in terms of programming difference:

<table>
<thead>
<tr>
<th>Model</th>
<th>Locality of data</th>
<th>Paradigm shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-world</td>
<td>Invisible (ideally)</td>
<td>Insignificant (ideally none)</td>
</tr>
<tr>
<td>Federated</td>
<td>Discoverable</td>
<td>Apparent</td>
</tr>
<tr>
<td>Confederated</td>
<td>Visible</td>
<td>Apparent</td>
</tr>
</tbody>
</table>

It is a responsibility of confederated implementations to make the transition from programming with local objects to programming with remote ones, as easy as possible. Whilst it is subjective to quantify the extent of a shift, we consider that as long as the following conditions prevail, the shift is acceptable:

- The granularity of data is the same in both local and remote operations. In other words if primitive values and objects are the units of granularity in local data manipulation, then a switch to say, raw bytes or pages is unacceptable.
- If local computation includes calls to routines, procedures or methods then the equivalent remote abstraction should also be an abstraction of similar semantics.

### 4.8. Summary

This chapter has reviewed various models of distributed persistence. Their weaknesses, particularly with respect to the properties of wide-area networks, have been highlighted. In view of these, a confederated arrangement has been advocated as being most resilient to wide-area networks. Of prime importance however, are the positive features present in these systems. The noteworthy features have been examined; they have been found to be desirable and should be incorporated into new confederated implementations.
Finally, the human factors of programming have been noted. Any programming activity that requires switching from one scenario to another, constitutes a shift in programming paradigm for the programmer. Here the shift occurs because the programming environment changes from manipulating local objects to remote ones. Ideally, a one-world model abstracts over this shift so that programming in a one-world distribution is no different from programming on a uni-processor system. However, in engineering terms, one-world models are only practical for small networks. In a wide-area network situation, the model chosen for the work described this thesis is the confederated model of distributed persistence.
Chapter 5
A Model of Confederated Persistent Object Stores

5.1. Introduction
The previous chapter advocated the practicality of the confederated persistence model. Among several systems, STACOS was examined and was deemed to offer the best model of distribution for wide-area distributed persistence. STACOS requires that programmers treat remote objects differently from local objects. To that effect though, operations on remote objects are fairly restricted. PJRMI was also found to be a good implementation. Unfortunately in PJRMI a remote reference that persists in a client causes the service at the server to be retained, creating permanent associations between them. This chapter now describes a model of a dynamic and scalable confederacy of persistent object stores. An earlier version of the model is described in [Lew & Brown 99].

We assume a number of independent stores, either as software stores in the style of the PJama Virtual Machine, or as the address space of a persistent operating system e.g. Grasshopper [Dearle et al. 93] or the memory of persistence-oriented hardware e.g. Monads [Rosenberg 90]. These stores would be geographically distributed, are likely to be owned by different organisations with different control and management policies and may well be implemented atop disparate hardware. These stores are willing to cooperate without compromising their autonomy and are connected to the wide-area network through which they communicate. This network is not trustworthy from both a communication reliability and from a hacker perspective. The Internet is a prime example of such a network.

5.2. Distribution and Failure Model
The diversity of existing distributed systems can be informally attributed to the following facts:

- different designers may have different goals for their system,
- the hardware platform on which they operate influences their decision,
- the operating system resources that allow them to exploit maximum benefits from the hardware and
- they have a specific end-user community in mind, whom they target for their system.

Above all, the distribution environment ultimately decides the architecture of a system if acceptable performance is to be achieved. The distribution environment of interest in this thesis is one where nodes only communicate via a network inter-connect. A node consists of one or more processors with a single level of memory, its persistent object store. The network may be a wide area or a local area network, or any combination thereof, linked by switches, routers and gateways. In such a landscape, nodal
communication takes substantially longer compared to intra-node data access. This in turn affects the cost of accessing data: accessing remote data is substantially more expensive than accessing local data.

Additionally, there is no central ownership or management of the entire network. It is made up of many organisations each of which owns part of it, so that the potential spread of the distribution spans many organisations. It follows that the technology used to implement the network is bound to differ across the distribution.

5.2.1. Influencing Factors

Given this state of affairs, this section identifies some factors that have influenced the design proposed later in the chapter. The factors are:

1. True distribution. By this we mean that the global space is not reliant upon any centralised resource or mechanism for its operation. The World-Wide Web is an example of a true distributed system.

2. Absolute scalability and beyond. Not only scalable, but dynamically evolvable—in other words, the ability to change the number of stores in the distribution without affecting the rest of the system while it still operates. This can be for any number of reasons, ranging from re-configuration to disagreement or policy changes between cooperating sites.

3. Site and organisational autonomy. Whilst many distributed systems cope with multiple sites, many of them ignore the reality of organisational barriers. This results in implementations where, to participate in the distribution, an organisation is required to provide access to its internal resources beyond a level that is deemed acceptable. For example, some systems such as MPI require login access on every machine used to run the parallel program. The provision of such support to outsiders is unacceptable to most organisations. The factor of influence here is that the network is fragmented from a geo-political point of view and the model must take such non-uniformity into account.

4. Nodal autonomy. A store rolls back if a local failure creates a need for reverting to the last stable state; never because another store has rolled back and affected it.

5. Security and minimal privileges. Following from 3 above, users are subject to security clearance by service providers. The autonomy requirement allows each store discretion to regulate the set of authorised users of its services. Depending on the granularity of control they want to maintain, a store may even have an authorised user group for each of the different services it provides. Minimum privileges mean that users are not granted more privileges than they need in order to obtain service. Ideally, users only need to connect and make requests. They need not specify how or when, neither should they be able to.

6. Fault-tolerance. Whilst it is possible to identify the potential sources of malfunction, deciding on a recovery action might amount to guess work. In this model therefore, the detection and flagging of abnormal situations is incorporated into the behaviour of the distribution. However, error recovery is left to the discretion of the application developer.

7. Hardware consists of a number of possibly heterogeneous computing elements connected by slow and unreliable networking. This, we believe, is currently the dominant feature of the Internet. By computing element we mean any computer ranging from the modest personal computer to expensive supercomputers.

8. The profile of a typical user is the application programmer who knows about the distribution of resources. This assumption is not unrealistic—nowadays people with little computer literacy do know about “Internet resources” and how to browse for them from all over the world. We further assume
that the application programmer who wants to access the resources, is aware that remote access is more expensive than local access. Therefore the programmer is willing to incur the costs of distribution, both in terms of programming paradigm shift and increased program execution time.

Given the distribution environment, the hardware profile and the organisational barriers assumed above, the table below summarises what is visible and what is transparent in our model:

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Visibility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locality</td>
<td>Visible</td>
<td>There is a distinction between what is remote and what is local. This conveys an awareness of cost to programmers.</td>
</tr>
<tr>
<td>Failure</td>
<td>Visible</td>
<td>An application should be allowed to decide its response if unable to access a remote object.</td>
</tr>
<tr>
<td>Protection</td>
<td>Visible</td>
<td>Programmers are aware that objects have access rights controlling them.</td>
</tr>
<tr>
<td>Security</td>
<td>Visible</td>
<td>Server nodes may authenticate clients. The choice of authentication mechanism is an individual responsibility.</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Transparent</td>
<td>Concurrent access to an object by programs in different parts of the distribution is managed by the host of the object.</td>
</tr>
<tr>
<td>Heterogeneity</td>
<td>Transparent</td>
<td>Programmers should not be influenced to develop their application by any difference in the hardware of nodes.</td>
</tr>
</tbody>
</table>

The primary need to make those visible characteristics clear to the programmer stems from the failure assumptions we make in our model.

5.2.2. Failure

Nodes in the distribution are autonomous and may fail without warning. So may the physical interconnect. In fact, the number of nodes is dynamic and varies as new nodes participate in the distribution and others fail or are cut off through link failure. A node that can be connected to, is deemed to be operational. By operational we mean that, subject to permission, resources hosted at that node are accessible to other nodes in the distribution. In other words, we assume it is not possible to connect to a host only to find that its persistent store is down. A node may crash, after which its persistent store will be unavailable until the node is up again. Despite crashes, all data present at the last stabilise will remain unaffected. This is an intrinsic property of persistent stores [Brown 88].

A more complicated failure is when physical breakdown of links causes the distributed architecture to partition into say, two disconnected worlds. Program executions that need to access resources in both partitions will at some stage find themselves unable to continue. If properly notified however, they might be able to seek alternative resources within their own partition and resume computation. It follows that there is a need for program executions to be given the option of reacting to failure. In order for this to happen, failure needs to be part of the distribution model. In view of this, the rest of this chapter develops a model of distribution suitable for confederated persistent systems.

5.3. Formative Concepts

The first concept introduced is that of a global space, the scope of which potentially spans the entire network. In this space, users may run computations that want to share resources. The space is made up of stores connected together in a virtual circuit arrangement. The collective operation of all the stores
A Model of Confederated Persistent Object Stores

constitutes the global space, in other words, the space is virtual—it does not exist by itself. Stores cooperate to share resources and interaction occurs in a secure way without being bound to any forms of permanent association. Figure 5-1 illustrates such a virtual global space with an arbitrary number of stores.

Each store supports a locality, which is a self-contained environment that is impermeable to direct references. That is, there is no operation defined on a locality that sends a local reference to a remote locality. In a locality, computations execute, potentially leaving behind objects that may persist. A locality may also offer a number of services that may be accessed by any computation in the global space. Localities are first-class objects and can therefore be nested. Those localities directly supported by the stores in the global space are top-level localities and they define the boundary of the underlying store.

Within a locality, a computation may directly access an object because both object and computation share the same space. Across boundaries however, computations may only request the use of a service via an indirection. An indirection is an object that identifies an entity hosted in another locality. A computation may obtain an indirection to a service in two ways:

1. by providing the name of the locality where the service is advertised, the name of the service and a signature identifying the type of the service, or,
2. by being sent an existing indirection from elsewhere. Indirections may be transmitted to a remote service to allow call-backs and third-party transfers. We say that a remote service performs a call-back when it uses an existing indirection to request resources from the original locality that sent it the indirection in the first place. A third-party transfer occurs when a locality A provides a locality B with an indirection to a service hosted in locality C. This causes more than two stores to cooperate towards solving a problem with each store contributing at least one thread of activity towards the computation.

Thus, given the ability to indirect to remote objects, computations are able to access the services pooled together by autonomous stores thereby achieving cooperation in a flexible way. We now examine these concepts in more detail.
5. A Model of Confederated Persistent Object Stores

5.3.1. The Global Virtual Space

The collective operation of all the stores in the confederacy constitutes a global virtual space in which computations run. No operation needs to be applied on the global space to include or exclude a store from the confederacy. Participation or withdrawal is solely a store-level decision and regardless of the outcome, the global space lives on until the last store shuts or breaks down. A store will typically be both a resource provider and a resource user. In other words, the global space is a peer-to-peer architecture.

5.3.2. Stores

A store is a node in the distribution. It is primarily a persistent address space that has absolute control over its closure, i.e. it can always be self-consistent. This space constitutes a named environment and the name uniquely identifies it in the global space. Stores also provide the necessary resources for computations to execute and to leave behind objects as their results. They manage their own object space by providing mechanisms such as garbage collection, stabilisation and recovery. Within the closure of a store, objects persist.

Figure 5-2 illustrates how a store in the global space, say store pos1.cs.adelaide.edu.au, can be visualised.

![Figure 5-2: A store named pos1.cs.adelaide.edu.au.](image)

Stores are stable and are able to evolve autonomously, without having to notify other stores. They evolve by stabilising their persistent address space regularly thereby protecting their population from failure. Any crash causes the store to revert back to its last stabilised state, effectively travelling back in time. Stores also start up and shutdown depending on local policy, and are not accountable to the rest of the virtual space. Each store has a persistence root, the closure of which, determines the persistent state of the store.

In this model the terms node and store are used interchangeably and we note that here, node does not necessarily mean a hardware node as in a processor board. Additionally, although the architecture is peer-to-peer, the term server is used to denote a node acting in its service-providing role whereas client is used for either a program or that part of a node that uses resources from servers.

Nodes can be any implementation of a computational environment that meets the following requirements:
1. provides support for the persistence of any object that a computation wants to retain, i.e. orthogonal persistence,
2. is stable and has the capacity to evolve,
3. can access network resources to communicate with remote sites, and,
4. is willing to participate by providing services to the global space and by satisfying incoming requests from the space.

For example, a node could be a single-user Napier88 store, a CASPER server or even a multicomputer-based single store image in the style of the MC-Texas store. As will be demonstrated in the later sections of this thesis, the stores used to implement a preliminary prototype of our model run on both single processor as well as dual processor shared memory machines. So in reality, our global space allows a mixed-mode model much in the same way as DPS-algol but without the pointer leakage.

5.3.3. Cooperation

Cooperation in this model is a two-pronged requirement to support distributed interaction. First, servers need to indicate their willingness to cooperate by making some of their resources available to the global space. This is termed participation in the global space and stores do so by registering resources. Registration is the act of converting a local resource into a service. In other words, when a node registers one of its objects, that object becomes a globally accessible service in the virtual space.

Second, clients cooperate by complying with the service access protocol in the space. They comply by indirection to services. Although the terms client and server are used to distinguish the service users from the providers, stores would usually assume both roles.

5.3.4. Localities

A locality is simply a self-contained scope in a store. It can be viewed as being a container in which objects can be placed to localise the persistent address space. Localities identify their contents by reachability and if they themselves are reachable from the store root, they persist. Localities are first-class objects and so, they can be nested. The persistent space of a store is a top-level locality although other mechanisms have been used in existing systems, for example in Napier88 it has traditionally been an environment [Dearle 89].

Within a locality, objects can refer to one another directly. Across localities however, relationships between objects are expressed using indirections. A locality does not share its objects with other localities. In addition to partitioning the persistent address space, localities are important because they are the containment mechanism we use to copy closures across stores. Figure 5-3 illustrates a store containing objects and localities.

5.3.5. Computations

A computation is the thread of activity arising from an application executing in a store where it uses resources and potentially modifies the state of the locality in which it executes. A computation is always initiated inside one and only one locality, its home locality.

A computation is a trusted activity in its home locality. Within the bounds of its home, a computation may directly access other residents, knowing that the access is fail-safe, and it may apply all the operations defined on the type of the resident being accessed. Across localities however, computations may only request the use of a remote object via an indirection. All operations to be carried out on the
remote object happen through the indirection and these operations are regulated by a generic call method defined over indirections.

Figure 5-3: Conceptual view of a locality.

5.3.6. Objects

An object is any unit of data or of activity that can be referenced using constructs available in the programming language used. It is the result of a computation, left behind in some locality of the store where the execution thread went through. It may be an abstraction of a real-life entity. An object may have a state and has an associated set of operations defined on it. A computation interacts with it via predefined operations either predefined by the language or by the programmer. Depending on the nature of the computation, an object can be a simple value, a data structure, another computation, a type, a procedure or an executable. For example the activity of the Napier88 compiler would generate new type representations, new procedures or new executables in its home store.

Our object model has been kept simple: objects typically have a state, part or all of which may be encapsulated, they are not mobile but easily communicate with one another, they are not replicated but nothing in the model prohibits implementing replication and they are local to a single store, their home store where they may persist. In STACOS, objects and their closures could be copied across stores. This was potentially expensive if a large closure, say PS, was copied. Here however, object-copying may only happen if the object type defines a copy, also known as clone in OO terminology, operation. By making it a programmer responsibility to define such an operation, without which copying is disallowed, potential "phantom-copying" is brought under control.

Objects also have an associated signature, which is their type representation. Most important of all, our objects are sufficiently fine-grained to represent the whole range of values and information structures that may be created using instantiation constructs available in modern programming languages. A signature is a representation of the semantics of, and operations defined on an object. Unlike type definitions, signatures are type identities that persist for as long as there exists an object of that type. Typically therefore, a signature would enable computations to ascertain the run-time type identity of an
5. A Model of Confederated Persistent Object Stores

object. Languages that support this concept have used various techniques ranging from unique patterns e.g. Modula-3's typecodes to actual objects e.g. Class objects in Java, and graph structures in Napier88.

Objects reside in localities and are directly accessible to computations executing locally. Before an object can be used from a remote store, it must be placed into the global space by its home store. **Registration** refers to the activity of making an object available for use by other stores in the global space. Registration may happen at any store wishing to contribute a service to the global community. The set of all registered services constitutes a global dynamic pool of shared reusable resources and every registered service **publishes** an interface, which essentially defines how remote requests are invoked.

The object model described here is very similar to the one from OO languages in chapter 3. This is desirable because it is important to keep the semantics of corresponding concepts as similar as possible regardless of where they are used. Failing to do so introduces potential paradigm shifts in the programming environment. Here the concept is that of an object that can either be applied to a single address space environment or to a distributed environment.

### 5.3.7. Services

A service is a resource that has been made available by a locality. Services are objects that have been registered in a locality for use by computations from outside. In other words, when an object is registered in a locality, it becomes the basis for some service that the locality wants to provide and is uniquely identifiable. We say that the object **implements** the service. For example, a locality intending to provide a random number generator to the global community does so by registering its number generator utility with a name and a signature. Thereafter, remote clients can indirect to the service to obtain random numbers. We note that in its capacity as a service, an object is completely encapsulated even if its type definition allows some state variables to be directly accessible. This complete encapsulation is enforced by the indirection mechanism which only allows clients to use a service by calling its operations. By the same token, "pointer leakage" is avoided. Detailed treatment of registration is given later.

### 5.4. Organisation of Objects in a Locality

Locality residents are organised by reachability. Typically an object becomes a resident when it is reachable from a root of persistence inside a locality. A locality can make some of its objects visible to the outside world. An object becomes visible by being registered as a service at a locality. At the time of registration, the object is given a name which can thereafter be used by remote computations to identify it. Not all residents of a locality are registered because some of them are private to the locality and only computations executing internally may access them.

### 5.4.1. Locality Structure and Operations

As mentioned above, a locality is a self-contained persistent space that identifies its object population by reachability. It follows that a locality either contains predefined root(s) of persistence or is able to dynamically create them when needed. A locality may also host computational threads of activities. Such local threads are encapsulated inside the locality; they do not directly refer to objects outside the locality boundary and objects created by local threads are instantiated using local resources, including local type information. Within a locality, threads may share objects because all the threads together with the objects are in the same space. Across localities however, indirections are used as explained below. Three major operations of localities are:
1. Listening and responding to requests for service.
2. Stabilising their persistent space at a convenient time so that their state can move from one consistent state to another.
3. Shutting down when instructed to do so by the node.

5.4.2. Service Registration
An object at a locality is said to be deployed when that locality makes the object globally accessible from outside. One way to deploy an object is through registration. Registration is an explicit activity whereby an object that would otherwise be invisible from outside a locality, is made visible to the outside world. It is important to distinguish between visibility and accessibility. Whilst visibility implies accessibility, the converse is not true; an accessible object needs not be visible.

The registration process gives a name to the object and once registered, the object becomes a service to the outside community. Despite making an object visible through naming, registration encapsulates the object so that external users interact with it only via operations defined on it. The operations defined on a service are those allowed by the object type.

Deployment can also happen without registration as will be examined below.

5.4.3. Propagation of Nested Registration
Localities are first-class objects in the persistent space. It follows that they can nest. In the event a nested locality inner with service s, is registered inside another locality outer, there is a need for s to be visible at the level of outer. Similarly, if at a later stage, outer becomes registered, its services—including s—become visible at the next locality boundary.

In this model therefore, nested services propagate to the outermost locality boundary wherever a registration path exists or is later created.

5.4.4. Stealth Objects
As explained above, a locality may make an object visible through registration. However, sometimes a locality may be reluctant to make an object visible, yet it still needs to allow access to that object. This is frequently the case when distributed computations cooperate using confidential data. In this model therefore, the threads inside a locality are able to deploy objects without registration. Such objects are called stealth objects. A stealth does not have a name and is therefore invisible. Yet the object is deployed and can only be accessed by an external client if the client is given an indirection and allowed to call back. It is only of use because somewhere in the global space, some computation will be provided with an indirection to the object and thereby be able to access it. Callbacks are explained in more detail later.

By the very nature of their use, these stealth objects are expected to be deployed as and when a thread inside a locality needs external cooperation. This is unlike services that are registered as and when the site owners believe they have a resource to offer for the external world to access at will. Therefore, this model stipulates that on-the-fly deployment is mandatory for stealth objects.

When a stealth is deployed, it becomes accessible. However, for external users to access it, they need some kind of mechanism to do so since the object cannot be seen. This is achieved through the use of an indirection: deployment of a stealth by a thread causes that thread to be given an indirection to the stealth. The thread is then free to distribute copies of that indirection to trusted external parties.
Callbacks

In a request to a remote service $S$, an application $A$ at some node $n$ may need to allow $S$ access to a local object $Obj$. Whilst call-by-copy is a common technique in existing remote call models, this is not always practical. Three examples where this impractical are:

- If $Obj$ is a large object or is an object with a large closure.
- In cases where call-by-copy semantics are inappropriate.
- If $Obj$ contains confidential data.

There is therefore a need for a mechanism to allow $S$ to contact $n$ back by indirecting to $Obj$. Callbacks are such a mechanism and are made possible through the use of a stealth object. A callback is made possible when $A$ instantiates an indirection to $Obj$ although $Obj$ is local and hands the indirection to $S$. Since $A$ and $Obj$ share the same address space, when $A$ instantiates the indirection it only needs to provide the indirection mechanism with a reference to $Obj$. In so doing, $A$ has effectively put out an unnamed object into the global space. However, because $Obj$ is only meant to be used by $S$, there is a need to restrict the visibility of $Obj$ from the rest of the global community—hence the need for $Obj$ to be unnamed. $Obj$ is a stealth object, invisible yet deployed in the global space. The flow of execution for a callback is shown in Figure 5-4.

![Figure 5-4: Execution flow in a callback.](image)

Stealth Removal

The primary objective of stealth objects is to allow safe access to resources that should not be visible yet need to be accessible to remote callers. It follows that a stealth object should be available at least for the duration of the callback. Unfortunately it is difficult to predict when a callback will happen. To address this uncertainty, it is a requirement of this model that a stealth object remains deployed for as long as the thread that deployed it is alive. This being a minimum requirement, an implementation may decide to promote the use of stealth objects beyond the lifespan of the deploying thread. In such circumstances, the implementation needs to decide when support for a stealth object may cease. The following implementation approaches are offered as a guide here:

1. One possibility is for a store to clear all its stealth objects when shutting down, or upon start-up if the last shutdown was due to a crash. It follows that with busy long-lived nodes, the amount of resources consumed just to support stealth interaction may exceed the level of commitment that the node owners are willing to provide. Therefore a more aggressive strategy may be needed.
5. Another possibility is for all stealth objects to carry an implicit timer, the expiration of which causes the object to be withdrawn from deployment. Association of a timer with a stealth object would be made when the object is deployed. Thereafter, every time the stealth is accessed by some remote client, its timer is refreshed. We believe this to be a sensible way in which to gauge the usefulness of stealth objects. This approach acknowledges the fact that some stealths may be more heavily used than others and that a single management policy may be undesirable.

3. And of course, the most restrictive yet most secure approach is to require that the computation responsible for putting out a stealth object also removes it when it terminates. This approach is least likely to open up a node to intruders in a hostile network environment.

Since stealth object management is an engineering issue, it is to be noted that our confederated model does not prescribe which approach is to be taken. Indeed each node in the confederacy may implement its own and node implementors are free to choose the approach that bests suits their site policy.

5.5. Indirections

An indirection is a local abstraction for a remote service and behaves like a network-wide typed reference to any object. Given two localities A and B, an application in A may refer to a service S hosted at B by instantiating an indirection to the remote service. Having done so, the application is thereafter able to request any operation defined by the service type to be carried out on S. To indirect to a remote service, a client would typically provide the name of the host locality, the name of the service as well as its signature.

Instantiation of an indirection establishes to the client, the fact that there was such a service available at the locality and that its signature matched. On the servicing side, it implies that the server has ascertained that the client knew the type of the service at the time of connection and therefore it may know how to call it.

5.5.1. Properties of Indirections

Indirections are self-contained and immutable. They preserve store autonomy whilst enabling interaction among cooperating and possibly heterogeneous stores. They present a type-secure interface through which interaction takes place and they have a set of exceptions that they may raise to signal various failures. These exceptions enable computations to detect failure and to take corrective measures.

Indirections are location-independent and may be transmitted across localities. Their mobility is the property that makes them most versatile in the global space. A locality may transmit several indirections in a remote call thereby allowing more than one store to participate in the computation, causing what we term, a computation spread. A computation spread is defined to be the collective activity of independent stores that results when an initial store sends out indirections to a number of remote stores, requesting their services in solving different aspects of a single problem. Effectively, each remote store contributes independent computations to solve the problem being addressed at the initial store.

Because indirections are small immutable objects, they may be efficiently transmitted to a remote site. Irrespective of where in the global space an indirection is, it always knows which service it is standing in for, which locality the service resides in, and how to communicate with it. Indirections may be instantiated for any object, regardless of type and of hardware heterogeneity.
5.5.2. Operations Defined over Indirections

The base confederated model as per Munro's work defined two operations over remote stores. They are browsing the content of a remote store and copying a remote object. Typically therefore, a computation would inspect the contents of a remote store to find some needed object; then a duplicate would be made and copied over for local use. The indirection mechanism defines additional operations:

1. constructors to instantiate indirections,
2. a clone operation for duplicating indirections,
3. an invoke operation which is essentially a generic remote call mechanism, and,
4. a query to interrogate the remote service and obtain a list of names of available operations.

Given an indirection to some service, invoke allows a computation to invoke the full range of operations defined by the type of the remote service. Query allows a computation to dynamically discover the operations supported by a remote service at the time of interacting with the service.

Semantics of Invoke

The main way whereby a client interacts with a service is by calling invoke on an indirection to the service. The call to invoke exhibits what is known as at-most-once semantics. In other words, either the operation request to the service is delivered and acted upon exactly once, or the request is guaranteed not to have reached the remote site, in which case service was not provided and the client is so informed.

Parameter Passing to Invoke

It is well-known in the literature that the two main parameter passing conventions are by-reference and by-value. As chapter 4 discussed, call-by-reference supports sharing semantics whilst call-by-value supports node autonomy. The chapter also examined distributed systems that only allow by-value transmission and saw that pointer leakage is prevented by copying all actual parameters, including their closures, across to the remote site. It was advocated that some computational models require geographically distant computations to share data. Both call-by-value and call-by-substitution fail to meet the requirement of these models. On the other hand by-reference semantics increase the coupling of nodes causing the distribution to become brittle. This too is unacceptable.

With the ability to pass indirections in invoke calls, a new convention in the form of call-by-indirection, is now available to distributed indirection-based computations. Where by-value calls are not adequate, through the passing of indirections to remote sites the effect of by-reference semantics can be achieved without pointer leakage.

In the context of parameter passing to invoke, the semantics are therefore as follows:

1. when a reference to a local object is passed as a parameter to an invoke call, by-value semantics apply. The object and its closure are copied across to the receiver of the invoke call.
2. When an indirectation to an object is passed as a parameter, the indirection is copied but because the indirection refers to an object, potentially any object of the global space, the indirection simulates by-reference semantics. We say "simulate" to distinguish from vanilla-flavoured call-by-reference where direct pointers are transmitted and the interacting nodes become dependent. No pointers are transmitted in call-by-indirection and the nodes retain their autonomy.

Therefore through the use of indirections as actual parameters, the restriction of calling by value only in confederated models is lifted.
5. A Model of Confederated Persistent Object Stores

**Invoke Failures**

In case a call to `invoke` fails, the model guarantees that the caller will be notified, although the latter may opt to ignore the notification. It follows that any error that potentially causes `invoke` to fail needs to be detectable. Section 5.6 examines error detection in our model.

5.5.3. Call-by-Indirection and Third-Party Interaction

With call-by-indirection, it becomes possible for store cooperation to go beyond the simple client-server interaction noted in section 5.3.3 above. Two possibilities have been identified and they are:

1. Callbacks. This has already been discussed above.
2. Third-party calls. This is a situation where three or more nodes participate in the distributed interaction.

A scenario of third-party interaction is now examined as an example. An application `A` requests a remote service `S` to perform some operation on a remote object `Obj`. Three separate stores are involved here: the store where `A` is running, the store that provides service `S` and the store that hosts `Obj`. `A` therefore instantiates two indirections, the first `I_s` is directed at `S` and the other, `I_obj`, at `Obj`. In its request to `S`, `A` then provides `S` with `I_obj` so that `S` may access `Obj`. The relationship between the stores is shown in Figure 5-5.

![Third-party interactions](image)

**Figure 5-5:** Third-party relationships

5.6. Error Detection

In most program runs, the success of the executing computation depends on timely and adequate availability of the resources needed by the run. Potentially, the more demanding the application is and the more diverse the resources needed, then the higher is the likelihood of one of them being unavailable. This may lead to a state where computation should not be allowed to proceed further. A trivial example is one where a program runs in a single address space and dynamically requests memory from its computing environment. If the request is denied and computation progresses, the thread of execution may corrupt its computational space.

In distributed situations, the problem is exacerbated because resource availability is typically supported by more than one provider. Their timely cooperation then becomes an additional determinant to the success of computation. Two main aspects are cause for concern in the confederated model. They are
communication failures and access violations. Our model therefore identifies a number of situations that are deemed to lead to resource unavailability. It is then possible to map these situations into exceptions and leave it up to applications to decide on the seriousness of the situation, e.g. if it is a mere network delay or whether it constitutes a failure. In languages that support exception handling this translates to a straightforward raising of throwing of the exception. In those that do not, error codes can be returned to the caller. These situations are examined next.

5.6.1. Communication Failures

The first potential source of failures are network hiccups, causing communication breakdown between the user of a service and its provider. This weakness is intrinsic to distributed systems and tends to be more apparent in wide-area arrangements. Node failures are also intrinsic to distributed systems given that such systems are composed of two or more interacting nodes. Communication failures are identifiable as one of three possibilities. They are:

- Whenever the client store attempts to connect to a known remote store that has shut down or failed, connection failure occurs. This failure is detected locally, can be mapped into a ConnectFailed exception which is thereafter thrown in the client application.
- Whilst the server store appears to exist yet, either: (i) it is not responding to requests from the client store, or (ii) the time that client store is willing to wait for a reply has lapsed. This situation is likely to arise with busy or geographically distant servers. TimedOut is the exception used to indicate this situation to the client application.
- NoRoute is used to flag network partitioning that causes the client store to be unable to find a route to the server store.

The three exceptions described above deal with store-level error situations. In other words, problems that arise between stores. These are the error conditions that arise at the lowest level in the model. We now examine error conditions that can arise at locality and request levels.

5.6.2. Access Violation

As is usual with operations defined on objects, the type of the object regulates access permission over the operations applicable to its instances. In both C++ and Java for example, member access specifiers in the form of keywords private, public and protected can be used to specify access rules.

Localities also enforce similar rules over services. Some operations defined by a service, may have been implemented for use by the service itself. When the service is deployed, these operations still form part of the service and if a remote computation knows the name of the private operation, it could still attempt to request its use. Exception AccessViolation is thrown if this situation arises.

5.6.3. Other Possible Failures

A client application instantiates an indirection to a remote service by providing the name of the service it wants to access and the name of the remote locality hosting that service. Exception NoSuchService is raised in the client application if no service of the given name exists in the remote locality at the time of the request.

The successful instantiation of an indirection by a client application merely establishes that there is such a service at the named remote locality. By no means does it guarantee that the client application will correctly use that indirection. Two factors govern this uncertainty:
5. A Model of Confederated Persistent Object Stores

1. being autonomous, the service provider might have changed the operations defined over the service,
2. the client may transmit the indirection to a third party who may have a different opinion of what service the indirection stands for.

Owing to the above, it is possible for an operation that is not defined over the service to be requested by the client. Additionally, even if a correct operation is requested, the wrong number or type of actual arguments may be supplied to the call. Our model therefore defines two exceptions, OperationUndefined and ArgumentMismatch, that may be raised in the caller to counteract such misbehaviour. If the caller is a third party service then the exception propagates to the original client. This propagation is analogous to exception re-throws in languages such as C++, where exception recovery can be distributed over several handlers.

In C++, a catch clause that first detects an exception may opt to do nothing or to only partly recover, and then re-throws, thereby leaving it to a different handler higher up the call stack to catch the re-thrown exception. The same concept is being used here, except that propagation transcends multiple address spaces.

In some implementations, store owners may wish to restrict access to services by regulating who is authorised to connect to their site. In such implementations, indirection instantiation may throw additional security related exceptions, e.g. UnauthorisedClient, should a client fail security clearance. Similarly, if a third-party does not have the requisite authority to request an operation, UnauthorisedClient could again be thrown.

5.7. Summary

This chapter has presented the design of an enhanced architecture for confederated persistent stores to extend Munro's original model. The new design adds flexibility by defining more operations over confederated stores. To support those additional operations, the locality concept has been introduced into the programming landscape and a new mechanism, called indirections, has been incorporated. Essentially, the mechanism allows applications in one store to refer to remote services via objects that behave like typed network-wide pointers to anything. Associated with the mechanism, is a set of error conditions that can be flagged to indicate the various failures that may happen in a network. Thus the new model features:

1. the concept of a locality as per DPS-algol, but one that does not allow pointers to leak,
2. programmer awareness of possible failure in any operation involving remote stores through the flagging of error conditions,
3. the preservation of referential integrity using indirections to remote objects and,
4. programmer-controlled copying of registered objects to avoid phantom copying.
Chapter 6
Implementing Indirections and Localities

6.1. Introduction
The first part of this chapter describes an implementation of the indirection concept presented in the preceding chapter. The assumed distribution is one where a node is a single store running on each host of the distribution. Each node listens to a socket at a certain port number. Nodes are multi-threaded so that in addition to servicing remote requests, they are also able to run a local application. In this implementation, a simple command-line interface was used to allow the user to interact with the node. The implementation is in Java/PJama. Java was chosen because of its extensive support for network programming.

After showing how indirections could be implemented, the chapter then briefly discusses how a confederacy of autonomous PJama stores can be built by simply connecting them in a virtual circuit and by making applications use indirections to interact with one another. The last part of the chapter uses PJama stores to illustrate one possible implementation of localities.

6.1.1. The Java Virtual Machine
Before jumping into the details of our implementation, a brief overview of the Java Virtual Machine, JVM, is appropriate. The JVM needed to run our implementation is just an abstract computing machine that follows the specifications laid out by Sun Microsystems [Lindholm & Yellin 97]. As chapter 7 will examine, we have used various JVM implementations for performance evaluation purposes. The relevant features of the JVM are:

- A run-time heap that is shared across all threads of activity. It is the runtime data area from which memory for all class instances and arrays is allocated.
- One or more threads of activity, executing in pseudo-concurrency. These threads independently execute code that access objects residing on the shared heap.
- Each thread has its own private stack, created at the same time as the thread. The stack of a thread stores frames—these are used to handle partial results, to perform dynamic linking, for return values and to dispatch exceptions. A new frame is created for every method invocation and is destroyed when the method completes.
- A garbage collector that automatically detects unreferenced objects, cleans them up and reclaims their memory space. We note the JVM Specifications do not require the garbage collector to ever strike during a particular instantiation of the JVM.
Typically, a JVM that supports orthogonal persistence would have more features. For example the PJama platform that we surveyed in chapter 2, is equipped with a persistent object store, the ability to perform atomic checkpoints, both explicitly by the programmer and implicitly when the JVM deems necessary as well as an intuitive API that is easy to understand and use.

Building on the above, our Java store provides a Store class which, when run on a JVM, causes it to be able to act as a node in a distributed environment. We now examine what is involved in making a JVM act as a confederacy node.

6.2. Indirections and Services

Figure 6-1 illustrates the organisation of our Java store and a node running an instance of such a store. Only those components that participate in distributed interaction are shown. Other typical components of the persistent store, namely persistence roots and garbage collector(s), have been omitted.

The following comments are relevant:

1. Each node listens to a TCP socket at port number 32767. We named this port the POS port.
2. At any time, only one local application may be running. The application may be multi-threaded. This is in line with current practice in most existing JVM implementations.
3. All deployed objects have an Object Indirection ID, O2D, that uniquely identifies them at their host node.
4. Clients that communicate with remote nodes may send two types of messages, initial messages and request messages. An initial message is sent to locate a remote service. It contains a Java String identifying the name of the service that the client believes to be deployed at the remote node. In
response, the server node replies with the O2D of the service—possibly zero if there is no such service. A request message is one where, having obtained a valid O2D the client now requests an operation from the remote service. The message contains the O2D, a Java String identifying the operation and an Object array of actual arguments.

5. Whenever a node accepts an initial message, it searches through its directory of services for an entry that matches the name in the message. If an entry is found, its O2D is transmitted back to the client else zero is transmitted and there is no further communication.

6. A stealth object is an object that has been implicitly deployed. It is not registered as a service and is therefore not reachable from the directory of services. Stealths are only accessible during callbacks. To enable a callback, a client instantiates an indirection to a local object and transmits that indirection to a remote node. The remote node is then able to call back. Instantiating an indirection to a local object implicitly deploys the object.

We define a class Store, an instance of which constitutes a node in the distribution. The main method of class Store is responsible for the following activities:

- Store.main starts by creating a daemon thread, the listener, that listens to a ServerSocket object at the POS port.
- Having done so, the main thread displays a command-line interface through which a local user may execute applications.
- Local applications execute in the main thread, henceforth called the local_run thread. The user enters the name of the class whose main method they wish to execute, together with command-line arguments. Store.main uses the facilities of Java’s core reflection package, java.lang.reflect, to dynamically load and execute the required application.
- Store.main maintains a directory of services together with an associative array of O2D-object pairs. All services have an O2D but not all O2Ds identify a service. A persistent global counter keeps track of the sequencing of unique O2Ds.
- Whenever the listener accepts an incoming connection from some client, it creates a remote_service thread to service the client. Typically the client would send an initial message and the new thread would read the service name contained, look it up in the services directory and reply with its O2D. Thereafter, the thread sits idle until it receives another message from the same site again. Subsequent messages need not be from the same client application, neither does it have to be for the same O2D—the only guarantee is that the message is from the same store.

### 6.2.1. Deploying Services

Class Store defines a public static method register that must be called to deploy a service in the store. Method register takes as arguments, an instance of the service and a Java String that is to identify the service. The signature of register is,

```java
static void register ( String the_name, Object the_service )
```

Calling register causes an instance of the_service to be entered in the Store’s directory of services under name the_name. An O2D is also issued for the service.

Therefore, deployment first requires the programmer to define a class, an instance of which will become the basis for the service. This activity is no different from what the programmer does in creating ordinary user-defined classes. Typically the class would also define a main method which when executed
at the node’s command-line interface, carries out the registration. All that is required of the main method is that it creates an instance of the class and invokes Store.register. The example in Figure 6-2 shows a code fragment of an image processor class. That class can be used to deploy an image processing service in a store. We note that there is nothing special with such classes; they are ordinary Java classes. It follows that class developers need not concern themselves with distribution aspects.

```java
public class ImageProcessor
{
    public int pixel_count ( PGM_image im, Integer threshold ) { // etc... }

    public int [ ] pixel_frequencies ( PGM_image im ) { // etc... }

    public PGM_image edge_detection ( PGM_image im, Integer threshold ) { // etc... }

    public PGM_image smooth ( PGM_image im, int [ ] weights ) { // etc... }

    public static void main ( String argv [ ] )
    {
        ImageProcessor ImProc = new ImageProcessor ( ) ;
        Store.register ( "Image Processor Service", ImProc ) ;
    }
}
```

Figure 6-2: A sample class implementing a service.

When class ImageProcessor is executed inside a store, a service named “Image Processor Service” is deployed in that store.

6.2.2. The Communication Layer

The communication layer is responsible for maintaining and recycling object streams among communicating stores. It manages the Socket, ObjectInputStream and ObjectOutputStream objects created by the listener in response to client connection requests.

An early implementation, developed as part of the work described here, opened a fresh Socket object every time a connection was needed between a client and a server store. Once the connection was established communication would occur, after which the socket would be closed. Whilst straightforward to implement, this approach proved inefficient in three ways:

1. Creating a Socket object has costly overheads, especially for simple communication such as indirection instantiation or when requesting fine-grained operations that perform little processing and require minimal data transfer.
2. On the server store’s side, for every new connection that is accepted, a new thread is created. When the client discards its Socket object, the server is left with a thread waiting on a dead socket.
3. All Socket objects linger for a while after closure. In programs that make a series of calls to remote stores, this has the effect of delaying the entire sequence of calls. Each subsequent call had to wait for the Socket object used in the previous call to close properly. In extreme cases, this could take up to 2 minutes.
The communication layer therefore implements a connections manager to re-use existing Socket objects. The manager abstracts over Socket specifics and acts as a communication medium that 'glues' cooperating nodes together. Every time a local thread needs to communicate with a remote site, a connection object is handed out by the manager. The connection object is given back to the layer when communication is done. A connection object is essentially a pair of object streams that allows two connected stores to send and receive Java objects. The manager is responsible for retaining connection objects.

The connections manager is implemented as a Tuple Space With A Difference, TSWAD. The only tuples stored in TSWAD are connection objects. Whenever a local thread does an in operation on TSWAD, the first tuple that matches the required remote destination is returned. If there is none, then in obtains a fresh Socket object, instantiates a connection using it and hands it to the requesting thread. When a thread is done using a connection object, it outs it back into TSWAD.

When a client instantiates an indirection to a service, the indirection constructor obtains a connection object from TSWAD and sends an initial message to the destination store. TSWAD is responsible for handing out a connection to the correct destination. As mentioned above, if there is none, TSWAD creates a new one to talk to the destination store. At the destination store, the listener accepts the connection and creates a daemon remote_service thread to service the new client. This thread is retained for further communication. On the other hand, if TSWAD found an existing connection object to the destination store, then the client bypasses the destination’s listener and is serviced by an existing thread.

6.2.3. Indirecting to Services and the Indirection Class

Given the name of a remote store and the name of a service deployed there, an application accesses the service by instantiating an indirection to the service. Thereafter, the indirection constitutes a handle on the service and can be used to request operations defined on the service. Class Indirection defines the operations needed to create and use indirection objects.

Constructors and Methods

Several constructors are defined by class Indirection, two of which are relevant for this part of the chapter. They are:

1. Indirection ( String remote_node, String service_name ), which is the primary constructor. It is used to instantiate an indirection to service_name deployed at remote_node.
2. Indirection ( Object stealth ), which is used to instantiate an indirection to a local object. The local object is implicitly registered and becomes a stealth. The indirection can thereafter be transmitted to other remote nodes.

Method clone ( ) is also available for duplicating an existing indirection. This operation is a local one in that it does not contact the remote store to obtain the O2D again.

Having instantiated indirections to services, applications may now ask them to perform operations. This is done using one of the invoking methods of class Indirection. The methods are:

- Object invoke ( String remote_operation ), used to request an operation that takes no argument and returns an Object result, which may be null. The operation being requested by this method is identified by its simple name and passed as argument remote_operation. This approach is analogous to that taken by Java's core reflection package, java.lang.reflect.
• Object invoke (String remote_operation, Object arg), used for invocations with arguments. If a single argument is to be provided then arg is a reference to the argument. For multi-argument invocations, arg references an array of objects. It is the caller's responsibility to organise the actual arguments into an Object array. We note that in Java an array is an object so that although arg is defined as a reference of class Object, an array can still be passed at invocation time. Method invoke checks the number of arguments passed by the caller.

The Indirection class also defines a query operation, String [] query (). Given an indirection to a service, when query is invoked, it returns a String array containing the simple names of all the operations defined over the type of the service.

Whilst two invoke methods are featured here, we note that this is primarily an implementation decision that was largely influenced by our use of Java. In a language like C++ where default argument values are supported, these methods could easily be collapsed into a single one.

Addressing Convention

In this implementation, the mechanism to uniquely identify a service uses a combination of the store's internet domain name and the service name. For example, say in store pos.adelaide.edu.au a service called "my_service" is registered. Given this, the following Java statement instantiates an indirection ind to "my_service":

```
Indirection ind = new Indirection ("pos.adelaide.edu.au", "my_service");
```

6.2.4. Characteristics of this Implementation

The most significant feature of the implementation described here, is its dynamic nature:

1. No client or server stub classes are relied upon. We decided against stubs because as is well-known in the persistence research community, the loading of classes into a persistent store impedes evolution unless versioning or evolution support is available. Without static stubs, upgrading a service only requires re-registering an instance of the new service with the same name. Thereafter, upgraded operations are effective from the next request onwards. Additionally, the use of static stubs makes on-the-fly deployment more difficult. Since the stealth mechanism is an important component of our model, it was important to show that an implementation could do without stubs.

2. Every indirection instantiation looks up the O2D of the service being indirected to. Persistent stores are dynamic and long-lived environments where client applications may outlive the services provided elsewhere in the confederacy. So whilst it would be trivial to cache O2Ds at the client store, we chose not to do so, in order to force a failure condition at indirection instantiation time. Otherwise, failure will arise at every subsequent operation request. In this implementation, failure to find an O2D is much cheaper than failure to obtain service. The latter activity involves building a signature for the operation before its non-existence is determined. In any case, should there be a need to create an indirection without a remote lookup, a call to Indirection.clone () can always be used.

3. The method implementing an operation is looked up every time it is requested. This decision is again influenced by the dynamic and long-lived nature of persistent stores. In a non-persistent environment where remote calls would usually be temporally close, it is unlikely that the operations defined on a service will change while clients are accessing the service. This assumption may break down in a persistent environment hence the desirability of dynamic binding and lookups.
4. Every remote operation undergoes dynamic type-checking. This is a rather expensive approach that resulted from using Java's core reflection package in order to discard static stub classes. This expense needs not be borne in other implementations.

In addition to the above, the following two situations are specific to our implementation. They are not prescribed by the model and other implementations may choose to differ.

1. An operation requested by a remote client may or may not execute in a newly created thread. Most requests received from the same client will be serviced by the same thread. Requests from different clients are guaranteed to be serviced in different threads. This mode of operation is very similar to that prescribed in Sun's Remote Method Invocation Specifications [Sun 97].

2. A service may execute multiple operations concurrently. This situation will arise if the server store has accepted requests for the same service from several clients. By the same token, multiple requests for the same operation may be executing concurrently on a service. This situation arises if the server store has accepted requests for some service from several clients and thereafter those clients all request the same operation from the service.

Chapter 7 will examine the costs involved in using indirections and identify where these costs are located. The next section shows how indirections could be used to organise PJama stores into a confederacy.

6.3. An Implementation of Confederated PJama Stores

Whilst the work described earlier did show that an implementation of the indirection mechanism is feasible, there is a need to now investigate how easy it is to incorporate the mechanism into existing persistence technology. This section briefly describes an experiment where we augmented PJama stores with the ability to handle indirections and produced a confederacy of stores. As will soon become clear, it was a trivial effort. The following subsections document the exercise and are necessarily brief since most of the implementation aspects have already been discussed.

6.3.1. Hardware Setup

The release of PJama that we used here only ran on Solaris platforms. To that effect we used three machines of the following configurations,

1. Dual Pentium II/300 with 512MB RAM and local disks, running Solaris 2.6/x86.,
2. Sun SparcStation Ultra 5/333 with local disks, running Sparc Solaris 2.6., and
3. Sun Enterprise 450 with four 300MHz UltraSparc processors and local disks running Solaris 2.6.,

in the networking arrangement shown in Figure 6-3.
6. Implementing Indirections and Localities

6.3.2. Store Structure

The structure of each node is not very different from that described in section 6.2. The only noteworthy differences are that:

- It is the PJama virtual machine that is being used to run a node.
- The services directory, the O2D generator and the index of services are persistent.
- When the node starts up, it verifies that it was last shutdown properly. If it wasn’t then it removes references to any stealth object from its index of services so that these may be garbage collected.

6.3.3. Creating a New Confederacy Node

To allow for the creation of stores that can participate in the confederacy, a simple utility program was written to create a PJama store file and to add the necessary data structures. The source code for the utility is as per Figure 6-4. Class Service implements a record-like object containing an empty tree of services, an empty array of references that eventually becomes the index of services and a counter, initially zero, for unique O2D generation.

```java
import org.opj.store.PJStore;
import org.opj.store.PJStoreImpl;
public class create_node {
    public static void main(String[] argv) throws Exception {
        PJStore ps = new PJStoreImpl(argv[0]);
        ps.newPRoot("Services", new Service());
        System.out.println("Created and initialised node...");
        System.out.println("Storefile is " + argv[0]);
        System.out.println("Services currently registered: " + s.count);
    }
}
```

Figure 6-4: Utility for creating a node.
6. Implementing Indirections and Localities

6.3.4. Running a Node

Once `create_node.main` has created a new store, the next step is to run a long-lived instance of the PJama virtual machine over that store. Again the preferred approach here is to present the local user with a command-line interface. Executing `Store.main`, the details of which were presented in section 6.2 is adequate for that purpose. Except for the following modifications at start-up, class `Store` is unchanged:

1. It now retrieves all its data structures from the persistent store.
2. It verifies if it was last shutdown properly.

We note that the above change in behaviour is necessary because of persistence. It is not a change in the distribution model.

6.4. Implementing Localities

One possible implementation of localities is now described. To be able to display all the properties explained in chapter 5—especially autonomy and impermeability—each locality is implemented as a Java object coupled with an instance of a nested PJama store.

As will be described later, class `Locality` is used to instantiate locality objects in our implementation. Each `Locality` object acts as a communication handle on the nested PJama virtual machine instance that provides the necessary persistent object space for that locality. Figure 6-5 illustrates the relationship between a `Locality` object and its backing virtual machine.

![Diagram of Locality object and virtual machine](Image)

**Figure 6-5:** `Locality` object plus backing virtual machine instance.
6.4.1. Organisation of PJama Store Files

Given that in a store node, there can be any number of localities nested inside the persistent store space, one can expect that there will be a number of autonomous virtual machine instances, the collective operation of which constitutes the computational capacity of the node. It also follows that there will be a corresponding number of PJama store files backing up that hierarchy of localities. This is shown in Figure 6-6.

![Figure 6-6: Directory of files backing up the persistent store space.](image)

The Top Level

The top-level is essentially unchanged from class Store as described in section 6.2, with the following two differences:

- There is now a port number dispenser, implemented as a daemon thread, that allocates the next free port number to every newly instantiated locality.
- Remote clients connecting to the top level may actually be looking for a service that is several levels of nesting deep. Therefore the listener thread at the top level has to track down the full path to the requested service. The listener thread no longer replies directly with the O2D of the requested service, rather, it is the nested locality hosting that service who is responsible for responding to the client.

6.4.2. The Locality Class

Locality objects are instantiated from class Locality, which provides the following constructors:

1. `Locality ( String ServiceName, String ClassName )` constructor where argument `ClassName` is the simple name of the class, an instance of which is to become a service inside the new locality. When this constructor is called, it uses the core reflection package `java.lang.reflect` to instantiate an object of class `ClassName`. This object is registered inside the newly created locality and is identified by the string `ServiceName`.

2. `Locality ( String ServiceName, String ClassName, Object ServiceArgument )` constructor. This constructor is similar to the first one but is used when the service to be instantiated needs an argument. In this case, `ServiceArgument` is passed to the constructor of `ClassName`.

3. `Locality ( String ServiceName, String ClassName, Object [ ] ServiceArguments )` constructor. This constructor is used when there are several arguments to the `ClassName` constructor.

When these constructors are called, they all create a `Locality` object in the caller, whose primary purpose is to act as a handle on an instance of a nested PJama virtual machine. This virtual machine instance is also created by the execution of the constructor call. The main fields of a `Locality` object are:
1. a Process object, the_jvm, representing the process that executes the nested virtual machine,
2. a pair of object I/O streams, to_jvm and from_jvm, opened through the standard input and standard output of the_jvm,
3. a port number, my_port, at which the Locality object listens for requests.
4. a String object, store_path—it is the filesystem directory path to the PJama store file for this locality.

Although the above constructors may seem restrictive in that they only create localities with single services, this can be overcome by making the initial service a “registering” service. That is, a service that supports the creation of new services when provided with class names and arguments.

Two other methods are defined by class Locality—a finaliser and a stabilise method. With respect to the finaliser however, it appears that with the newer versions of the Java Virtual Machine, it is not possible to force the execution of finalisers on virtual machine exit. Method stabilise when invoked on a Locality object, instructs its nested virtual machine instance to invoke the static method stabilizeAll of class PJStoreImpl.

Locality Instantiation—the Algorithm

As mentioned earlier, a Locality object is created with an embedded service inside it. Localities are instantiated through a call to new, with appropriate arguments as described above. When such a call is made, not only is a Locality object created in the caller but there is also a need to start a PJama virtual machine instance to provide resources for the service to operate. The steps towards creating an operational locality are as follows:

1. One of the Locality constructors described above is called, to create a Locality object in the caller. This constructor also creates a Process object that starts a new instance of the PJama virtual machine.
2. When the virtual machine starts up, it creates a new PJama store file—henceforth, this store file is the persistent object space of the new locality. That PJama store file is placed in the hierarchy of store files as previously described—see Figure 6-6.
3. The virtual machine then reads in all the information needed to instantiate a new service. This information is provided by the Locality constructor—essentially the information is the service class, the service name and any arguments needed by the service constructor.
4. Once the service is created, it is registered—it is almost ready to be used. Before this can happen, the new locality needs to know the port number where it must listen, in order to accept requests.
5. The virtual machine connects to the top-level locality of the node and obtains the next available port number. Having done this, the locality now has all the necessary information for autonomous operation. A commit operation ensures this information is made persistent.
6. Finally, a thread is started and made to listen on a ServerSocket. Henceforth, a remote service thread is created every time a new connection is accepted from that ServerSocket.

In addition to the above, a thread is also needed to maintain communication with the immediate parent of this locality. For example, when the entire node shuts down, its localities shutdown in a recursive fashion—i.e. before a locality shuts down, it is responsible for instructing its children to shutdown first. This thread is the main thread of the nested virtual machine and the communication takes the form of the redirected standard I/O—System.in and System.out—through which object streams have been opened.

In the current implementation of localities, the above steps are carried out by two classes: createNestedVM and runNestedVM. Class createNestedVM carries out steps 1 to 5 and terminates whilst
runNestedVM is responsible for step 6 and executes for as long as the node is operational. This approach was deemed necessary because of a quirk in the PJama implementation used. In the current PJama implementation, a virtual machine instance that creates a new PJama store file may only stabilise implicitly when the virtual machine instance exits—that is, an explicit stabilise is not allowed. Therefore, in the present implementation of Locality constructors, the Process object first runs a nested virtual machine instance to execute createNestedVM.main. When createNestedVM.main terminates normally, step 5 would have been completed. The Process object then executes runNestedVM.main—this execution lasts until the node shuts down.

Thread runNestedVM.main

As mentioned above, the nested virtual machine instance of a locality object needs to maintain communication with its creator. This need is justified in the following two situations:

1. A computation may pass a Locality object as an argument to an Indirection constructor. For example:

   // Assume a locality object imageProcessorLocality is in scope here. Now we want to
   // indirect to service "ImageProcessor" offered by locality imageProcessorLocality.
   // So we instantiate an Indirection as follows:
   Indirection ind = new Indirection ( imageProcessorLocality, "ImageProcessor" ) ;

   In the above instantiation, the Indirection constructor requests the O2D of service "ImageProcessor" from the locality object imageProcessorLocality. Communication between imageProcessorLocality and its virtual machine instance happens through the redirected System.in and System.out.

2. A Locality object sometimes sends instructions to its virtual machine instance, for example when the locality performs its stabilise method or when there is a request to shut down as explained earlier.

Deep-Copying of Localities

No clone () method is defined by class Locality. Rather, class Store defines a static method duplicate that accepts as argument a Locality object. We note that this distinction is important because if class Locality were to define clone (), it would mean every locality object may be cloned. However, the ability to do so would depend on the content of the locality. Some localities may contain objects that refer to external entities, e.g. Process, File and Socket objects, and it may not make sense to duplicate them.

The following approach was therefore deemed more suitable: when Store.duplicate is called it attempts to make a replica of its argument. Because a locality may contain an object whose cloning is prohibited, the method may throw a CloneNotSupportedException. Duplication will only be successful if all the classes of the objects contained in the locality implement interface Cloneable.

Locality Addressing

When taking localities into consideration, the mechanism to uniquely identify a service now uses a combination of the store's internet domain name and the full path to the service. Paths are specified from the top-level locality of a store and through nested localities, if any. For example, say in store pos.adelaide.edu.au there is a nested locality "my_locality" registered and in it, service "my_service" is registered. Given this, the following Java statement instantiates an indirection ind to "my_service":

   Indirection ind = new Indirection ( "pos.adelaide.edu.au", "my_locality::my_service" ) ;
6. Implementing Indirections and Localities

Note that "::" is used as a locality separator in the style of the C++ scope operator.

**Indirecting to Non-Registered Localities**

Nested localities that are not registered can be indrected to, in the same fashion as with stealth objects. In particular, the indirection constructor: Indirection ( Locality a_locality, String service_name ) is to be used. Figure 6-7 shows an example of how this is done:

```java
// Say we have a nested locality containing an ImageProcessor service called
// my_processor as instantiated on the next line.
Locality a_loc = new Locality ( "ImageProcessor", "my_processor" );

// So, we instantiate an indirection to my_processor.
Indirection improc = new Indirection ( a_loc, "my_processor" );

// Now invoke an operation, eg "detect_edge" on some image.
// Build the arguments for the operation first and then make the invocation.
Object [] args = { new PGM_image ( "an_image.pgm" ), new Integer ( 150 ) };
PGM_image edges = (PGM_image) improc.invoke ( "detect_edge", args );
```

Figure 6-7: Indirecting to non-registered nested services.

This indirection constructor is also useful in creating stealth objects out of embedded services. If there is a non-registered nested locality in some store, the services deployed in the nested locality are only visible to computations that can directly reference that locality. Should there be a need for any of its services to be confidentially accessible to remote computations, a local computation may create an indirection to the service and transmit it to remote localities.

**6.4.3. Miscellaneous Optimisations**

The above sections described one possible implementation of localities. Depending on the actual distributed environment at hand and the assumptions that can be made of users, a number of optimisations can be made. These optimisations ultimately depends on the compromises that implementors are willing to make. Invariably, all optimisations imply some trade-offs. Two small suggestions are briefly covered next.

**Service Mirroring**

Since services may be cloned and transmitted across localities, it is possible for site owners to agree on replicating services to maximise availability. Say some service S at some locality L is mirrored as service S clone at locality L'. We also assume that it would be desirable for users to access the locality that, in terms of network distance, is closest to them. Given this scenario, one possible optimisation would be for the cooperating localities to hand out the O2D + IP address of either S or S clone depending on which one is in closer proximity to the client.
Signature Verification

If a client instantiates an indirection to a service at some remote locality, then whenever the client requests an operation, the operation name and its actual arguments are transmitted to the remote locality. There, the argument locality types are used to verify that such an operation is defined on the service. If none is found, the remote locality returns a SignatureMismatchException to the client. It follows that in the case of signature mismatch, not only was the server’s compute resources unnecessarily used up but network bandwidth was also wasted.

To minimise this wastage, it is possible to have signature verification performed at the client. Not only does such local signature matching save on network transmission but it also distributes the verification process to clients on a per-request basis thereby saving server resources for potentially valid requests. To support local verification, the client needs to be provided with the signature of an operation sometime before the first request for that operation. Conveniently, this can be provided at the time when the client instantiates an indirection to the remote service. In reply to the client’s lookup message for the service, the remote locality transmits the service’s O2D. In addition, the signatures of all operations on the service are sent to the client. Thereafter, the client is equipped with all the necessary information for signature verification.

Unfortunately, this optimisation would only be worthwhile in a landscape where client nodes have adequate processing power and client-server distances are large. Some local area networks are typified by a few powerful compute servers and many relatively weak client machines. In this case, it may be more efficient for clients to constantly communicate with and depend on servers for all processing, including the type checking.

Also, this optimisation requires a trusted client store. We also note that this is an optimisation that would only speed up error reporting.

6.5. Implementation Hiccups

The following implementation difficulties have been encountered and are largely influenced by the way Java and in particular, the PJama implementation operates:

1. When the locality constructor creates a PJama store, it cannot explicitly call method PJStoreImpl.stabilizeAll. This is just the way the current PJama implementation operates. It follows that after a locality is instantiated and its embedded service created, any attempt by the service constructor to call stabilizeAll will break the locality. If there is a need to avoid this, the locality constructor can be made to immediately shut down the newly created store and then re-open it through a second call to Runtime.exec. Thereafter, the service constructor can be called.

2. Because a nested locality is backed by a full-blown virtual machine instance running in its own right, it loads the same implementation of system and run-time classes as the top-level instance. Thus nothing prevents a service thread from making a call to static method System.exit. Such a call would cause the virtual machine instance to exit, leaving a locality object without a backing virtual machine instance. To address this problem, services should not have access to System.exit.

3. When there are no more references to a nested locality object in the persistent store, the object is garbage collected. However the Java Virtual Machine may or may not invoke the finalizer before the locality object is reclaimed by the collector. If it does not, then the store file on which the nested locality relied is not removed from the filesystem. This results in a mismatch between the directory of store files and the logical hierarchy of localities making up the node’s persistent space.
4. Using several virtual machine instances to implement nested localities inevitably means that more resources are needed to support a store containing many localities. The tests conducted so far indicate that the highest costs incurred are at store start-up, where one virtual machine instance needs to be executed for each nested locality. On a Sun UltraSparc 5/333, an average cost of 2 seconds is incurred per additional virtual machine start-up.

With respect to 4 above, there is a need to note that once the entire store has started up and all the localities are operational, response time then depends very much on the underlying hardware architecture. On single processor machines, no parallelism is available and all the virtual machine instances compete for a single CPU. In a busy store, it is therefore expected that the presence of many localities would be reflected by low performance. In a symmetric multi-processing environment however, the situation is likely to be different because more processors are available to concurrently run the many virtual machine instances of the store.

To verify the above, we ran a node on a Sun Enterprise 450 equipped with 4 processors and observed the behaviour of the virtual machine instances. Using output obtained from a utility program called top, we noted that the virtual machine instances did run concurrently as and when clients requested service. We believe that the observation supports the expectation noted in the above paragraph.

6.6. Summary

This chapter has discussed a preliminary but complete implementation of indirections and localities. The possibility of using indirections to arrange autonomous PJama stores as a confederacy was also shown to be feasible. It is important to note that, except for reflection, the implementation did not rely on any support that is not commonly available in modern programming languages such as Java.
Chapter 7
Performance and Assessment

7.1. Introduction
To gain acceptance, a communication mechanism should be at least as viable as existing technology. To that effect, this chapter is mainly concerned with performance issues. The chapter first discusses what test scenarios were used to obtain insight into the performance of Indirections. Graphs and charts obtained from preliminary performance figures are also presented and finally, a qualitative look is offered on the paradigm shift involved when programmers use indirection and locality objects in their code.

7.2. Indirections—A Quantitative Evaluation
As may be inferred from the model developed in chapter 5, there are three situations where the performance of indirections as a communication mechanism needs to be ascertained. They are,

1. In a simple interaction scenario where a client communicates with a remote service provider.
2. In third party situations where a client exploits services from more than one remote server.
3. In the presence of callbacks where the remote server needs resources held at the client site. Here, the performance evaluation implicitly tests the usefulness of stealth objects.

A popular mechanism that Java programmers may use for distributed programming, is Java’s own Remote Method Invocation, RMI. The performance tests described in the remaining part of this chapter have therefore compared Indirections with Java’s RMI. In absolute terms, there is no universally agreed approach on how to objectively assess distributed computing performance, especially not in a way that is independent of network topology. By choosing a comparative approach with RMI, it is hoped that programmers who are familiar with RMI will be able to gauge the relative performance of indirections.

Even with the comparative approach though, the network influences the outcome. The next section therefore describes the physical network used.

7.2.1. Hardware Setup
The various platforms we have experimented with are a combination of DEC AlphaStations running Digital Unix 4.0, a dual Pentium II/300 running Solaris 2.6, a 133MHz Pentium PC running Windows NT4.0 Workstation and two Sun UltraSparc IIs with dual 170MHz UltraSparc CPUs running Solaris 2.5.

At the time of conducting the tests, the RMI implementation available on the DEC AlphaStations was inefficient and performed poorly. Therefore, all the results obtained from experiments involving an
AlphaStation have been disregarded. The relevant section of the network used for the rest is as pictured in Figure 7-1.

Not all the machines ran the same version of Java. The NT workstation ran IBM's Developer Kit Java Edition 1.1.7, the dual Pentium II ran Sun's JDK 1.2 and the UltraSparcs ran Sun's JDK 1.1.4. However, we were consistent during testing and we were careful to use identical set-ups when comparing RMI and indirections.

![Network Diagram]

Figure 7-1: Relevant hardware and operating systems arrangement.

### 7.2.2. Test Programs and Overall Results

Java programs to test the three types of distributed interactions described above were written as explained next. We relied on the following two scenarios to write test programs: (i) distributed image processing and (ii) distributed parts manufacturing with a single serial number generator. Performance measurements were made and their results are commented on in the rest of this chapter. For all of them, measurements were averaged over at least 15 runs.

#### Simple Interaction—Image Processor Example

In this test case, the client application creates an image object, then accesses the server to request some operations to be performed on the image. The client requests four operations from the server, they are:

(i) **pixel counting**—this operation returns the number of pixels above a given threshold. The image and the threshold are passed as actual arguments to the remote call.

(ii) **histogram**—this operation returns an array of 256 integers representing the frequency distribution of each pixel value in the image. The image is an actual argument to the call.

(iii) **edge detection**—given an integer threshold and an image as actual arguments, this operation detects edges in the original image and returns a new image containing the edges.

(iv) **a smoothing algorithm**—given an image and an array of 9 integer weights as actual arguments, this operation returns a new smoothed image.
In the RMI version, a remote interface `ImageProcessor` defines four remote methods: `pixel_count`, `histogram`, `detect_edge` and `smooth`. Class server implements `ImageProcessor` and its `main` method simply binds a server object named “ImageProcessor” to the RMI registry. The client starts by creating its image object from some PGM file [DEC98]. It then looks up the server for remote object “ImageProcessor” via a call to `Naming.lookup ( )`. The lookup returns a reference to “ImageProcessor” and with the reference, the client then invokes the four remote methods.

For the indirection-based version, class `ImageProcessor` defines the four methods needed, as shown in Figure 7-2. A similar version was presented earlier as Figure 6-2.

```java
public class ImageProcessor
{
    public int pixel_count ( PGM_image im, Integer threshold ) { // etc... }

    public int [] histogram ( PGM_image im ) { // etc... }

    public PGM_image detect_edge ( PGM_image im, Integer threshold ) { // etc... }

    public PGM_image smooth ( PGM_image im, int [ ] weights ) { // etc... }

    public static void main ( String [ ] argv )
    {
        ImageProcessor ImProc = new ImageProcessor ( );
        Store.register ( "Image Processor", ImProc );
    }
}
```

**Figure 7-2:** Image processor source code.

The server is a store that registers an `ImageProcessor` object and waits for requests. To register the service the above class is executed inside the server store. The client creates an image object from some PGM file and instantiates an indirection to the image processing service. Thereafter the generic method `invoke` defined by class `Indirection`, is used to successively request the four methods of the class `ImageProcessor` to operate on the image object. The client program is as shown in Figure 7-3. It accepts two command-line arguments: the IP address of the machine hosting the image processor and the filename of an image in PGM format.

The graph in Figure 7-4 compares the overall performance of the two versions. Image files of various sizes, ranging from 1KB to 1.3MB were used. Results have been averaged over 25 runs. All timing values were obtained using calls to the method `System.currentTimeMillis ( )` at the start and end of the `try..catch` block. In the hardware set-up for this particular test, the client ran cold from the NT4.0 workstation, requesting services from a hot server on the dual Pentium II running Solaris 2.6. The results obtained from the other Solaris hosts indicated similar trends.
public class client  
{
    public static void main ( String [ ] argv ) throws UnknownHostException, IOException
    {
        PGM_image im = new PGM_image ( argv [ 1 ] ); // argv[1] is image file.
        Object [ ] arg1 = { im, new Integer ( 150 ) };
        int [ ] wts = { 60, 5, 5, 5, 5, 5, 5, 5, 5 };  
        Object [ ] arg2 = { im, wts };
        try
        {
            Indirection ImProc = new Indirection ( argv [ 0 ], "Image Processor" );
            Integer val = ( Integer ) ImProc.invoke ( "pixel_count", arg1 );
            int [ ] hist = ( int [ ] ) ImProc.invoke ( "histogram", im );
            PGM_image edges = ( PGM_image ) ImProc.invoke ( "detect_edge", arg1 );
            PGM_image smoothed = ( PGM_image ) ImProc.invoke ( "smooth", arg2 );
        }
        catch ( IndirectionException e ) { System.out.println ( "client: " + e ); }
    }
}

Figure 7-3: Client source code.

![Graph showing RMI vs. Indirections in simple interaction.](image-url)  

Figure 7-4: RMI vs. Indirections in simple interaction.
Third-Party Calls—Image Processing Example

In this set-up, three hosts are involved. One UltraSparc II acts as a repository of data, the dual Pentium II is a compute server and the client runs on the NT workstation. Again a distributed image processing scenario, shown in Figure 7-5, was used.

In this set-up, three hosts are involved. One UltraSparc II acts as a repository of data, the dual Pentium II is a compute server and the client runs on the NT workstation. Again a distributed image processing scenario, shown in Figure 7-5, was used.

In this particular set-up, the client is a trivial application that makes use of resources available at the other two sites. In the RMI version, the image repository implements an interface called remote_image and makes available a number of images that can be accessed through remote_image references. The image processor service implements four remote methods that accept a remote_image reference as one of their arguments. They are again pixel_count, histogram, detect_edge and smooth.

Given an image name as a command-line argument, the client looks the image up and obtains a remote reference to it. It then looks up the image processor and again obtains its remote reference. Thereafter, the client makes calls to the four remote methods, and passes the image remote reference as an argument.

In the indirection-based version, the repository is a store in which image objects have been registered and given a name. The image processor is also a service on a different host. The client, on yet another host runs from a cold virtual machine. The client is as coded in Figure 7-6 and the graph in Figure 7-7 compares the performances of the RMI and indirection versions. Garbage collection activities in both experiments were taken into account. Essentially, all the virtual machines ran with the verbose:gc switch turned on and this caused the time spent on every strike of the garbage collector to be printed out. This time was deducted from the overall execution time of the run when collection occurred.
public class thirdparty
{
    public static void main ( String [ ] argv ) throws UnknownHostException, IOException
    {
        long end, start = System.currentTimeMillis ( );
        Indirection im = new Indirection ( "krypton", argv[0] ) ;
        Indirection p = new Indirection ( "inverness", "Image Processor" ) ;
        Object [ ] args = { im, new Integer ( 150 ) } ;
        int [ ] wt = { 60, 5, 5, 5, 5, 5, 5, 5, 5 } ;
        Object [ ] args2 = { im, wt } ;
        PGM_image result = (PGM_image) p.invoke ( "smooth", args2 ) ;
        Integer val = (Integer) p.invoke ( "pixel_count", args ) ;
        int [ ] frequency = (int [ ]) p.invoke ( "histogram", im ) ;
        PGM_image edges = (PGM_image) p.invoke ( "detect_edge", args ) ;
        end = System.currentTimeMillis( ) ;
        System.out.println ( end - start ) ;
    }
}

Figure 7-6: Third-party client source code.

![Graph](image)

Figure 7-7: Third-party performance comparison.

**Callbacks and Stealths—Parts Factory Scenario**

A callback situation is one where for some reason the client is unwilling to transmit all the information needed for the remote operation through by-value arguments. It therefore passes as an argument to the remote operation, an indirection to an object in the client’s store. Using this indirection argument, the server store is then able to call back.
When using RMI, programmers typically have a waiting server, bound to an RMI registry somewhere, and a number of clients looking up the registry to locate a bound object. Therefore, callbacks are not common in RMI because this would require clients to be bound and be waiting as well. It is possible however, to have two interacting servers bound to two different instances of the RMI registry on different machines. To test callbacks, we used the following scenario:

1. A client needs to access a remote factory of parts to request the manufacture of, say a car door.
2. Car doors, like all engineered parts, have a unique serial number issued to them upon manufacture.
3. Only the client may generate a serial number for parts it requests from the factory.
4. The client is unwilling to generate serial numbers prior to the call to the factory in case its request to the factory is denied and the serial number is wasted.

The client therefore instantiates an indirection to its number generator thereby registering it as a stealth object in its own store. It then instantiates an indirection to the factory and in its request to the factory for a car door, it gives the number generator indirection to the factory. Thereafter, the factory calls back, is issued a serial number for the car door that it manufactured and returns the completed door to the client. The indirection-based version of this program is fairly straightforward. The factory is the class, shown in Figure 7-8, which when run in the server store, registers an instance of itself. The client code is presented in Figure 7-9. The following comments are relevant:

1. When the client creates an indirection to a local object—here the number generator, this implicitly registers the local object as a stealth.
2. A door object is a simple object with two fields, a String description of the door, e.g. "car door" and an int serial number.

```java
class factory {
    public static door make_door ( Indirection num_gen ) throws IOException {
        Integer num = null;
        try {
            num = (Integer) num_gen.invoke ( "next_serial_number" ) ;
        }
        catch ( Exception e ) { System.err.println ( "manufacturer: "+ e ) ; }
        return new door ( new String ( "car door" ), num.intValue ( ) ) ;
    }

    public Object clone ( ) throws CloneNotSupportedException {
        return super.clone ( ) ;
    }

    public static void main ( String [ ] argv ) throws NoSuchMethodException {
        Store.register ( "factory", new factory ( ) ) ;
    }
}
```

Figure 7-8: Factory service code.
public class client
{
    public static void main ( String [] argv ) throws IOException, ClassNotFoundException
    {
        long start = System.currentTimeMillis ( ), end ;
        NumGen g = new NumGen ( ) ; // Instantiate local
        // number generator.
        Indirection serial_num_generator = new Indirection ( g ) ; // Stealth creation.
        Indirection parts_maker = new Indirection ( argv[0], "factory" ) ; // argv[0] is the host.
        for ( int i = 0 ; i < Integer.parseInt ( argv[1]) ; i++ ) // argv[1] is number of
            // calls to make.
            door d = (door) parts_maker.invoke ( "make_door", serial_num_generator ) ;
        System.out.println ( d.descrip + " " + d.serial_num ) ;
    }
    end = System.currentTimeMillis ( ) ;
    System.out.println ( end - start ) ;
}

Figure 7-9: Client code for stealth/callback.

The RMI version involves two servers on different hosts. One server implements the factory and the second one implements the number generator. The second server looks up the factory and then calls remote method make_door. Method make_door accepts a remote number generator argument. The hosts used in this experiment are the two Ultra Sparc II/170s.

Figure 7-10: RMI callbacks vs. Indirections callbacks.

Though Figure 7-10 shows that callbacks using indirections are much more cost-effective than in RMI, we need to note that applications using indirection callbacks may only do so when running inside a store. So whilst the test results discussed earlier in this chapter always involved cold clients, in this case the clients are running in a store environment where the necessary classes and threads are already loaded in the virtual machine. This would account for much, though not all, of the performance gain over RMI. Other reasons for such a wide performance gap are as follows:
even when cold, indirection instantiation is much cheaper than remote object lookup. A breakdown of costs is given later.

To allow the server to call back, the client makes a call to `java.Naming.bind` in order to register the local object with the RMI registry. In real time, this is a very expensive operation to perform on-the-fly. Having done so, the client then looks up the object to obtain a remote reference. This requires yet another call to `java.Naming.lookup` before the client can finally proceed with its request for invocation from the server. In contrast, an indirection-based callback only requires the client to instantiate an `Indirection` to the local object. The `Indirection (Object stealth)` constructor takes care of deployment.

**Fine-grained Interactions**

So far, results show that the indirection mechanism performs well in distributed applications involving medium to coarse-grained data transfers and remote operations. The situation may be totally different when it comes to simpler and finer-grained interaction. Therefore in a separate experiment RMI was compared with indirections in applications that require minimal data transfers and computation.

Figure 7-11 charts the relative performances of RMI and indirections when a client accesses a remote number generator to obtain a serial number. Computation at the server is minimal and involved incrementing a static integer and returning its value to the client. The call from the client to the server is also simpler in that it takes no actual arguments. The two hosts used were the Ultra Sparc II/170s.

![Figure 7-11: Relative performances in fine-grained interactions.](image)

In Figure 7-11, the gradient of the indirection line is actually steeper than that of RMI. This can be explained by the fact that because of its static nature, RMI actually features cheaper remote calls. However, the instantiation cost of remote references is high. This is unfortunate because one frequent scenario in distributed applications is where the client is weak in processing power compared to the server. Having the client incur a high instantiation cost can deter programmers from using RMI in such situations. We now examine individual costs.

**Cost Identification**

In the particular settings used earlier, i.e. image processing and parts manufacture, indirections have proved to be affordable. However, there is a need to identify where the costs are centred. In both RMI and...
indirections, two programming activities consume most of the execution time. In the RMI case, they are (i) remote object lookup and (ii) remote method invocation, whereas in the indirection case, they are (i) indirect instantiation and (ii) request for a remote operation. The use of the terms “method” with respect to RMI as opposed to “operation” with respect to indirections is purely philosophical—whilst RMI is specific to Java, indirections are a language independent concept and the generic term “operation” is preferred.

Instantiation and Call Costs
To gain deeper insight into the proportion of costs of each of the above components, we have timed the components in a separate image processing experiment. Using a single image of 100KB, ten remote calls to the same operation, namely edge detection, are made after the initial lookup or instantiation. The hosts involved are the NT4.0 workstation and the dual Pentium II/300.

The graph in Figure 7-12 individualises the costs incurred by lookup/instantiation and each of the subsequent remote calls. For the indirections tests, we have used two clients, one that ran on a cold virtual machine and the other in store where the classes needed by the indirection mechanism have been pre-loaded. The first two columns of each group in the chart compares RMI with cold indirections. This chart is representative of all the tests conducted on the various hardware combinations mentioned earlier using different image sizes. We observed:

1. The cost of RMI’s remote object lookup to be roughly twice that of instantiating an indirection in a cold client. This observation takes into account the fact that our network is a local one—the mileage may vary over long haul arrangements.
2. Both RMI’s first call and indirection’s first request to be always higher than subsequent ones.
3. That after the first call or request, subsequent ones were of the same costs, with RMI calls cheaper than indirection-based ones. Maybe because the same method is called over and over again with the same arguments, RMI performs some optimisation. In a normal application, such repeated calls may not happen.
4. That the first request for a remote operation was always substantially higher cost than the rest.

![Figure 7-12: Breakdown of costs of instantiation and invocations.](image-url)
Observation 4 can be explained by the fact that some classes, namely those from the `java.lang.reflect` package, are being loaded for the first time, both at the client's and the server's virtual machines. We are not worried by this seemingly higher initial cost because as mentioned earlier, indirections are a mechanism targeted for persistent store environments where these classes would have already been loaded. The third column in each group of the above chart confirms that we can afford not to worry. As expected, in a total store environment, the first operation request costs the same as subsequent ones. Instantiating an indirection is also a lot cheaper because not only are all necessary classes loaded but idle `remote_service` threads at the server are likely to have already been created.

To understand why a remote operation request is more expensive than a remote method invocation, we need to consider the sequence of activities that happens when a client makes such a request. For example, for edge detection on an image, the Java client code to make the request is:

```java
PGM_image edges = ( PGM_image ) ImProc.invoke ( "detect_edge", args );
```

where `ImProc` is an indirection to the image processing service, `args` is an `Object` array with the original `PGM_image` object and the `Integer` threshold as zeroth and first elements. The corresponding RMI call is:

```java
PGM_image edges = ImProc.detect_edge ( im, threshold );
```

where `ImProc` is a reference to the remote server that implements the image processor interface and `im` is the original image object. The `threshold` argument is the cut-off value at which a pixel is deemed to be part of an edge.

In the indirection-based request, method `invoke` obtains a `connection` object to the remote host, sends a message containing the O2D of the current indirection `ImProc`, string “detect_edge” and the arguments array `args`. `invoke` then blocks, waiting for the result. At the remote site, the servicing store tracks down the image processing service using the O2D embedded in the message. The types of the arguments provided in the message are then determined. This type information, in combination with the string name of the operation “detect_edge”, is used to invoke method `detect_edge` of the image processor object. Finding operation “detect_edge” involves the use of the `java.lang.reflect` package. When the method returns, its result is sent back to the client store. At the client store, method `invoke` returns in the client and the result is cast into a `PGM_image` object. This casting again undergoes a type-check.

In this experiment, apart from instantiation and the first invocation, an analysis of the numeric values of invocation costs indicates that for the network setup we used, the following relationships apply:

- Invoking an operation via indirection is 30% more expensive than the corresponding RMI call. This is largely due to the use of the `java.lang.reflect` package for tracking down the operation to be invoked as well as the use of dynamic type checking of actual parameters. The latter is examined in detail below.
- The 30% extra-cost mentioned above is only about 25% of the total cost of an indirection-based `invoke`. Over a wide area network with significant communication latency, the extra cost incurred by indirections may become negligible.

### Parameter Checking Costs

As is clear by now, one fundamental difference between RMI and indirections is the dynamic nature of indirections. RMI generates client stub and server skeleton classes which, at compile time, are statically checked and bound to the client and the server.
In a persistent environment, this static binding is undesirable and therefore the indirection mechanism features dynamic type checking. At the time of servicing a request for a remote operation, the server store builds an array of class `Class` objects, corresponding to the actual argument list. This array is then used to find the method implementing the remote operation. Finding the method uses Java’s core reflection package and involves dynamic type checking and class loading if necessary. Once the checking and loading are done the method is then called, knowing that the actual arguments are of the correct type.

All this array building and checking at run time can be expensive. We expect this to be more pronounced in operations performing little computation and returning small-sized results but with long argument lists. Figure 7-13 charts cost against increasing number of arguments. Instantiation is excluded because only call costs are of interest.

![Figure 7-13: Cost comparison with increasing number of arguments.](image)

In the above experiment, the methods did no processing and returned no result. All the arguments were also small—subclasses of `Object` with no fields or methods of their own. As suspected, Indirection’s dynamic type-checking causes costs to increase with higher numbers of call arguments.

### 7.3. On Confederated PJama Performance

An extensive performance assessment of indirections in a confederated PJama environment was not carried out because initial testing showed that as far as the trends of the graphs are concerned, little had changed compared to the results already presented. However, it is to be noted that PJama’s RMI, PJRMI, appears to be closer in performance to indirections. Two plausible explanations are:

1. The version of PJama used is based on Sun’s recently released JDK1.2. In this release, RMI is known to perform better than previous JDK1.1.x versions.
2. A PJama store incorporates an RMI registry as an integral part of its virtual machine. There is no need for a separate registry and hence, no inter-process communication between store and registry.
A glimpse into relative performance between the performance of indirections and PJRMI can be obtained from Figure 7-14. In that case, three PJama stores were involved in a third-party interaction. Only one remote operation, namely edge detection, was carried out. The hardware setup used is described in Figure 6-3 and was made up of a dual Pentium II/300, an Ultra Sparc 5/333 and an Enterprise 450/300 all of which ran Solaris 2.6.

![Figure 7-14: PJRMI vs. Indirection.](image)

### 7.4. Impact of Nested Localities on Performance

The previous section showed that Indirections and PJRMI are comparable in performance, with Indirections being slightly more expensive overall although this depends on the number of remote operations invoked.

When localities are introduced as a software composition tool, the difference in cost between Indirections and PJRMI becomes more pronounced—with Indirections being higher in cost by at least 25%. This is because, in addition to the two reasons given in section 7.3 above, the following is to be considered:

1. All initial messages to a nested locality go through the top-level locality of a node. The top-level locality creates a thread to track down the nested locality before forwarding the message. In other words, Indirection instantiation becomes more expensive because of this three-way communication.
2. When the nested locality receives a message from the top-level, it creates a new thread again, to service the request.

### 7.4.1. Locality Instantiation Cost

As explained in chapter 6, the instantiation of a locality includes the following steps:

- Starting up a PJama virtual machine to execute `createNestedVM.main` to create a new PJama store.
- Invoke the service class constructor using Java's core reflection package and register the resulting object as a service in the locality.
- Exiting the PJama virtual machine, thus causing the new store to stabilise.
Starting yet another instance of the PJama virtual machine using the newly created store file with its persistent service. This is the machine instance that executes `runNestedVM.main`.

On a Sun UltraSpare 5/333, all this may take in excess of 10 seconds. However, it is to be noted that most of this cost is due to the need to create two instances of the PJama virtual machine. Hopefully, with future releases of PJama, a newly created store will be able to explicitly stabilise and continue execution. Currently, a virtual machine instance that creates a new store may not stabilise explicitly.

### 7.5. A Qualitative Look

This section qualitatively evaluates Indirections and Localities. As would be the case with most examinations of this nature, subjectivity is very much part of the conclusion. For any programming construct to be useable, it eventually boils down to two main criteria. They are:

- What kind of performance can be expected. This is implementation dependent and has already been dealt with.
- How easy it is to program with. This aspect, together with paradigm shift is discussed next. Here paradigm shift will be examined as a comparison between the distributed setting and the uniprocessor environment.

There are three areas where a paradigm shift might be present and where the programmer could be faced with difficulty. They are:

1. Writing code that implements service classes.
2. Instantiating indirections and using them to make remote calls.
3. Organising software into localities.

The next three sub-sections will now examine each one of them.

#### 7.5.1. Implementing Services

As was pointed out earlier, any object can be deployed as a service through registration at a node. No special programming is needed to code a class that is used to instantiate the object and the same class can be used for a uniprocessor environment. Therefore no paradigm shift is observed here. For the distributed case, once the object is instantiated, registration needs to be explicitly made through a call to the node's `register` method. This step is simple and is no different from method invocation on a uniprocessor—again there is no paradigm shift.

With RMI however, there are differences in the way ordinary classes are developed as opposed to those classes whose instances are to be bound to an RMI registry:

1. Interfaces are first defined. These are needed so clients can interact with remote RMI objects.
2. The classes whose instances will be used as RMI remote objects are required to implement the interfaces mentioned in 1 above, as well as extend some system-provided classes—usually `java.rmi.server.RemoteObject` or `java.rmi.server.UnicastRemoteObject`.
3. Finally, a special compiler is typically used to generate stubs.

Having done so, the step to make an object remotely available is now straightforward: a call to the method `java.rmi.Naming.bind(String hostName, Remote theObject)` binds the object to the registry. The object is thereafter ready for remote invocation.
7.5.2. Programming with Indirections

In a uniprocessor environment, a reference to an object can be obtained by simple definition; to create the object, an instantiation is required. In Java for example, the following code defines a reference to an ImageProcessor object, and the actual object is created through a call to new:

\[
\text{ImageProcessor im\_proc = new ImageProcessor ( ) ;}
\]

Thereafter, methods defined or inherited by class ImageProcessor can be invoked, with zero or more actual arguments. For example, to invoke detect\_edge which takes an image and an integer threshold as arguments:

\[
\begin{align*}
\text{PGM\_image im} & = \text{new PGM\_image ( "an\_image\_file.pgm" ) ;} \\
\text{PGM\_image edges} & = \text{im\_proc.detect\_edge ( im, 150 ) ;}
\end{align*}
\]

When indirections are used to access non-local objects, programmers are presented with a dichotomy in the object world: the local ones, manipulated via direct references and the remote ones that can only be indirected to. When instantiating an indirection to a remote service, one of the arguments that the programmer provides, is the name of the host where the service is located. For example:

\[
\begin{align*}
\text{Indirection im\_proc} & = \text{new Indirection ( "ahost.com.au", "the\_image\_processor" ) ;}
\end{align*}
\]

Having done so, the programmer can thereafter use method Indirection.invoke to request operations:

\[
\begin{align*}
\text{PGM\_image im} & = \text{new PGM\_image ( "an\_image.pgm" ) ;} \\
\text{Integer threshold} & = \text{new Integer ( 150 ) ;} \\
\text{Object [ ] args} & = [ \text{im, threshold } ] ; \\
\text{PGM\_image edges} & = (\text{PGM\_image}) \text{im\_proc.invoke ( "detect\_edge", args ) ;}
\end{align*}
\]

As is obvious from the code fragment above, Indirection.invoke may require that arguments to the remote operation be organised as an Object array. This approach was borrowed from Java’s core reflection package java.lang.reflect.

Therefore, two related but different programming skills are needed when programming with indirections. They are:

1. An understanding of distributed systems such as the Web and knowledge of Internet host names.
2. Awareness of the difference between objects that are local and those that are remote, together with the appreciation that remote access is typically more expensive, less reliable and that the access mechanism is different.

In addition, familiarity with linguistic reflection, at least to the level offered by java.lang.reflect, is helpful.

As explained in the previous chapter, two simpler versions of Indirection.invoke are also available for the following cases:

1. When the remote operation requires no arguments.
2. When a single argument is to be transmitted.
In those two cases, no Object array needs to be created, making the programming exercise no more difficult than on a uniprocessor.

When programming with RMI references, apart from the initial lookup, there is little difference from programming on a uniprocessor. The following code fragment shows the initial lookup of an image processor, of class RemoteProcessor, bound to a remote RMI registry, together with a call to one of its methods, detect_edge.

```java
RemoteProcessor im_proc = (RemoteProcessor) Java.rmi.Naming.lookup
    ( "//URL.address somewheren" + "the_image_processor" );

// Henceforth, invoke methods in the usual, distribution-transparent way.
PGM_image im = new PGM_image ( "an_image.pgm" );
PGM_image edges = im_proc.detect_edge ( im, 150 );
```

7.5.3. Using Localities to Build Components

The main operation provided by class Locality is instantiation through one of its constructors. When a locality is instantiated, an embedded service is also created. To create the embedded service, the service class needs to be provided so that the service constructor can be looked up by reflection. Once this constructor is obtained, an object is instantiated using it. The signature for a service constructor is one of three possibilities:

1. One that takes no arguments, i.e. the no-arg constructor in Java terminology.
2. One that takes a single argument.
3. One that takes more than one arguments.

In cases 1 and 2 above, the necessary programming skill is no different from that needed to create any Java object through a call to new. The relevant Locality constructors are Locality ( String ServiceName, String ServiceClass ) and Locality ( String ServiceName, String ClassName, Object ServiceArgument ).

With the third case, coding is more involved because of the need to organise all the arguments into an Object array—this requires at least one extra line of code. Constructor Locality ( String ServiceName, String ClassName, Object [ ] ServiceArguments ) is used in this case; for example, say the service constructor needs two arguments of class Class1 and Class2 respectively:

```java
Object [ ] argv = { new Class1 ( some_arguments ), new Class2 ( more_arguments ) };
Locality a_locality = new Locality ( "my_service", "a_class_name", argv );
```

Here again there is a departure from the traditional “list of arguments” that is usually passed during calls. Whether this is a considerable programming paradigm shift or not, will depend on the programmer. Programmers who regularly use Java’s core reflection package java.lang.reflect will find themselves in familiar territory.

With the use of localities as a component-building tool, programmers also need to be aware that references do not cross locality boundaries. This awareness is important when writing the code for the methods that will be called on embedded services—any assignment to a seemingly “global variable” will cause the locality to create a local copy of the global variable. For example, in a node where the persistent space is made up of localities, even the root(s) of persistence are no longer global. Each locality, including
the top-level, has its own set of roots and a computation executing inside one locality is only able to access the roots of that locality.

7.6. Summary

This chapter has examined the performance of the implementation described in chapter 6 compared to Java's Remote Method Invocation, RMI, mechanism and showed that indirection-based computing is affordable. Assessment focused on the three main ways in which distributed programs may interact, namely simple two-way interaction, third-parties and callbacks. A breakdown of invocation costs was also presented. It was noted that the experiment was conducted on a local area network, the topology of which would invariably influence the test outcome. This simple comparative approach was adopted because of the inherent difficulty in measuring distributed systems performance.

Section 7.3 of the chapter showed that Indirections were affordable when used to organise PJama stores into a confederacy. Section 7.4 explained the overheads incurred when programming with nested localities and the final part presented a qualitative assessment of the paradigm shift when using indirections and localities.
Chapter 8
Software Architecture and Re-use

8.1. Introduction
The ability to distribute a persistent space over a confederacy of stores enables a judicious organisation of resources for composing software into autonomous units. The goal of this chapter is to propose a model of software structuring using localities as composition units and indirections as the “glue” holding the units. The advantage of such an approach is that the resulting architecture exhibits several desirable properties such as autonomy, modular decomposition, abstraction and information hiding, fault-tolerance and support for software evolution. These properties have also been documented in [Lew & Brown 99a].

The justification for advocating the above approach is as follows. A number of applications are sufficiently complex, both by their sheer size and in the functionality they support, to warrant a design that divides the functionality among separate components. In other words, we are effectively decomposing the application into a number of logical, self-contained and well-behaved units. Typically, this type of application is dynamic, making a monolithic structure undesirable. Even when organised into components, a flexible and resilient medium is needed to inter-connect the components in a way that supports the dynamic nature of the application.

In single persistent environments, hyper-programming [Kirby 92] is a choice mechanism for developing such applications. Often though, the size and requirements of such applications call for large-scale processing and storage capacity of the type normally available in distributed environments. Yet hyper-programming is still new technology and in its current form, it does not translate well to a distributed environment.

8.2. Kirby’s Work on Hyper-Programming
Hyper-programming is the activity that composes a program using source code, augmented with direct references to items in a persistent object store. The items can be either data or, more interestingly, code. It follows that a hyper-program is a program that is able to discover new code from its run-time environment and incorporate that code into its behaviour. Such programs display a number of advantages over conventional ones. Four of them are:

- A hyper-program tends to be more succinct in its composed form.
- A hyper-program is able to achieve true code sharing, that is, without ever copying from program libraries.
• Well-defined software composition is possible—an aspect of persistent software development that is of prime interest here.
• The links embedded in a hyper-program can be checked for correctness early during development, typically at the time of composition.

Clearly, hyper-programming affords increased programmer expressiveness and with the benefit of succinctness, there is a strong case for hyper-programming in large-scale software composition. For additional benefits of hyper-programming, the reader is invited to consult [Kirby 92].

8.2.1. Limitations of Hyper-Programs

One of the requirements of a hyper-programming environment is that the environment provides a hyper-link mechanism. In current hyper-programming systems the tendency has been to assume a single persistent store, so that hyper-links could then be implemented using direct references or pointers. The use of such references makes the objects "pointed to" reachable and included in the program closure. Unfortunately large applications often make use of distributed resources. In its current form, the hyper-link faces three main obstacles that hamper the development of distributed hyper-programming systems. They are:

1. Raw pointers are rarely meaningful across address spaces.
2. Tight coupling results from the use of such pointers, making store management difficult. This is exacerbated in distributed environments with multiple stores.
3. It can be difficult to upgrade/substitute components that make up a hyper-program, especially when a number of links weave in and out of the components.

Another related and implicit assumption is that the program under construction together with the items being linked to, and the resulting executable, all reside in a persistent store. This is only the case in sophisticated systems like Napier88 [Dearle et al 89]. Other than that, the majority of persistent systems, especially the C++ derivatives, are more primitive. They still rely on traditional files to store code and executable, whilst persistent data reside in a persistent store.

The survey of existing persistent systems in chapter 4 indicated a concerted effort by the research community to formulate distributed persistence models. In particular, this thesis has favoured the confederated persistence model as first synthesised by Munro in the STACOS project. Therefore, in line with the observation that persistent stores have adapted to distribution, this chapter envisages that eventually, hyper-programs could also span wide-areas.

8.2.2. Towards Distributed Hyper-Programs

For hyper-programs to span multiple stores, the following minimum requirements are identified:

1. An addressing mechanism that is similar to hyper-links; but one that can cross address spaces. Confederated stores cooperate over wide areas and the mechanism must therefore survive the unpredictable conditions that may plague such an environment.
2. A reliable communication medium, preferably with "at-most-once" semantics, to connect the store nodes. The medium should also be able to flag failures or other exceptional circumstances so the hyper-program can self-correct.
3. A component structuring mechanism to organise the hyper-program into well-defined fragments, each of which may be hosted at a different persistent store.
In addition, tools are needed to help programmers understand the semantics of data hosted at remote persistent stores. Remote browsing technology has already been well researched [Munro 93] and incorporated into some systems, e.g. Napier88—therefore this requirement is not addressed here.

In a confederacy, persistent stores are autonomous and so the linking mechanism needed must be one that does not increase coupling among them. That is, a store that allows external clients to link to its objects does not become bound to any permanent association. By the same token, a client that is allowed to link to a remote object may never assume the latter will always be available—hence the need for the communication medium to be able to report unexpected circumstances. And finally, the need for a structuring mechanism is not only for organising the application into well-defined components but also to allow each component to decide which objects are visible and can be remotely linked to, and which ones are for internal use only.

8.2.3. Hyper-Worlds

To address the upgrade limitation mentioned in section 8.2.1 above, Kirby proposed the hyper-world concept. An illustration of a persistent store with hyper-worlds, reproduced from [Kirby 92], is shown in Figure 8-1. Essentially a hyper-world is a persistent store entity that contains the program and data components of an application, together with type descriptions. Each hyper-world would make visible a single component to which other parts of the application may link. By so structuring applications in the persistent store, the latter effectively becomes a collection of hyper-worlds. This potentially eases store management by imposing a coarser granularity on persistent store entities and by restricting hyper-links to a single entry point per hyper-world.

![Figure 8-1: Kirby's representation of a persistent store with hyper-worlds.](image)

The following observations can be made:

1. hyper-worlds appear to have well-defined entry points through which interaction from outside can happen.
2. hyper-worlds can be nested to impose finer structuring over the relationships among entities of the persistent store.
3. that the inter-connect relating hyper-worlds to other entities are hyper-links, the implementation of which would still be based on raw pointers,
4. and that from within a hyper-world, these same pointers are allowed to refer to the outside world.
5. The locality concept, as described in chapter 5, exhibits similar properties to 1 and 2 above.

Observations 3 and 4 seem to indicate that hyper-worlds, if implemented, were primarily meant for use in a single persistent store environment. However, by replacing direct pointers with indirections and using localities instead of hyper-worlds, this powerful composition approach can be retained and adapted for the distributed case, through the use of localities. In addition to easing manageability, autonomy would also be preserved.

### 8.3. Software Composition with Localities and Indirections

Having advocated the desirability of well-structured persistent software components in the preceding discussion, this section now examines how localities can be used as the basis for such software composition. Indirections are then used as the inter-connect “glue” for locality interaction. In other words, we advocate that localities and indirections are appropriate tools for building a complex and potentially distributed application.

The model of software composition proposed here is one where:

- An application is a complex piece of software that consists of a number of programming objects each of which displays a range of operations. The term *functionality* will be used to refer to such “ranges of operations”.
- The functionality of the application is well-defined so that the application can be structured into several self-contained components.
- The self-contained components interact with one another in a well-behaved manner and the collective operation of the components constitute the overall application behaviour.
- The application is based inside a persistent environment, where all its resource requirements can be met by the environment. This is very similar to Napier88 programs.
- Resources needed by the application may or may not be distributed—that is, the persistent space may consist of several persistent object stores. However, regardless of whether resources are distributed or not, the application structure is unaffected.

Given the above application model, there is a need for programming tools that support the composition of persistent software of the kind conceptualised above. Localities and indirections adequately meet such needs in that:

1. Localities can be used to organise the application functionality.
2. Indirections will provide the medium through which well-behaved interaction happens.

#### 8.3.1. Composing the Building Blocks

The use of localities to structure a complex application into a number of components entails that each locality encapsulates an application component. That is, each component would be based inside a locality and the operations defined on that component would be the operations defined on the locality’s embedded service. In the Java/PJama programming context, ideally each component would be defined by a class.
Then for each such class, a locality is instantiated using one of the three *Locality* constructors described in chapter 6.

### 8.3.2. Connecting the Components

Having decomposed the application into locality-based components, there is now a need to provide a communication medium via which the localities interact. As advocated, the indirection concept provides such a mechanism. Using any of the *Indirection* constructors, the programmer is now able to instantiate indirections to the components and request the necessary computation according to the purpose of the application. To illustrate this we now re-examine the third-party image processing example of chapter 7.

### 8.3.3. Image Processing Example Revisited

Chapter 7 introduced an image processing application example where a client creates indirections to a remote image and to a remote image processor. The client then requests the remote image processor to perform some operations, e.g. edge detection, on the remote image. Diagrammatically, this structure is as shown in Figure 8-2.

![Figure 8-2: Distributed image processing application structure.](image)

Whilst in the above scenario, the three localities are remote with respect to one another, it is not difficult to see that in terms of the application structure, little would change if nested localities were used. Figure 8-3 shows the nested case.
In Figure 8-3 above, the objects repository and imageProcessor are of class Locality. Inspection of the application code further confirms that little has changed. Figure 8-4a presents the code for the distributed case whilst Figure 8-4b is the nested case, with the differences highlighted in bold.

```java
public class thirdparty
{
    Indirection img = new Indirection("repository.com.au", "image1");
    Indirection improc = new Indirection("processor.com.au", "ImageProcessor");
    Object[] arg = { img, new Integer(150) };
    PGM_image edges = (PGM_image) improc.invoke("detect_edge", args);
}
```

**Figure 8-4a:** Source code for distributed third-party interaction.

```java
public class thirdparty
{
    // Assume existence of locality objects repository and imageProcessor.
    Indirection img = new Indirection(repository, "image1");
    Indirection improc = new Indirection(imageProcessor, "ImageProcessor");
    Object[] arg = { img, new Integer(150) };
    PGM_image edges = (PGM_image) improc.invoke("detect_edge", args);
}
```

**Figure 8-4b:** Source code for nested third-party interaction.
8.4. Trade-off Between Autonomy and Static Binding

This section now discusses one important factor when replacing pointer-based hyper-links with indirections. In a single persistent store, once a hyper-link has been included into a source program, the programmer can rest assured that,

1. The association between the hyper-program and the object stays valid for as long as the hyper-program needs the object.
2. The type of the object being linked into the source matches the context in which the object is being used, and cannot be changed.

Static binding therefore affords the ability to perform program checking at an early time and is regarded as an important benefit of hyper-programming. We concur. Yet whilst the ability to do so has long been a safety feature of hyper-programming, we suggest that there is an inherent tension between store autonomy and static binding. The reason is because composition-time checking relies on the assumption that the programmer knows the path to a persistent object and is able to bind the object to the hyper-program via a token. Having done so, reachability guarantees that the bound object will still be available when the hyper-program executable eventually runs.

In a confederacy however, the situation is different:

1. Store boundaries are visible and reachability does not usually span multiple stores.
2. Stores are autonomous and a store gives no indication to others when it is about to discard an object that used to be available for remote use.
3. Network link failure may make a remote object inaccessible despite its availability at the remote store.

In such an environment therefore, it is not sufficient to rely solely on composition-time checking because thereafter, a node may discard an object that is remotely needed, thereby invalidating the check. However, composition-time checking is still valuable because it guides the programmer through the construction of a correct program. Subsequent checks are necessary to guarantee correct program behaviour since the initial composition-time checks would only have restricted failures to events beyond the programmer’s control. These checks must occur at run-time but could also be performed at compile time to provide additional confidence. The run-time checks would typically happen at the time of dynamically binding to remote objects and clients would rely on exceptions to detect failures.

Notwithstanding the above, there are some cases where run-time checks are unnecessary. For example, some small confederacies are centrally managed and the nodes trust one another. It is then possible to implement indirections that behave like direct pointers, much in the style of PJRMI’s remote references. We note however that in PJRMI, a remote reference that persists causes the remote object to persist in the remote store as well. Over time, the remote store administrators may find it difficult to decide which objects can be safely unbound from the RMI registry.

8.5. Code Re-use

A major difference between a locality and a hyper-world, is the ability of the locality to completely shield objects from direct referencing. Using localities therefore, it now becomes possible to give out fragments of an application as persistent components that are self-sufficient. For example, a development platform vendor could implement a software development environment as a number of self-contained tools, each of
which is based inside a locality. Customers would initially purchase a first implementation of the environment but thereafter they would be able to upgrade specific fragments. By acquiring newer releases in locality-increments they would have the discretion of upgrading only those fragments that are really important. At the moment, this is not possible with current hyper-programs because of the presence of direct references.

8.6. Summary

This chapter has proposed a model of software composition in a potentially distributed persistent environment. The type of software that this model targets is fairly specific—the model is appropriate for applications that are complex and whose functionality can be decomposed into well-defined units of behaviour. Given the need to develop such an application, localities can therefore be used as the basis for the well-defined components. These components in turn constitute the building blocks of the application and the indirection mechanism is the glue that connects all units together into a resilient inter-connect.
Chapter 9
Conclusion

This thesis has investigated the persistence abstraction with respect to object-oriented languages and in the context of a persistent object store that may distribute its computation to other cooperating stores if needed. The work described here started as an experiment to determine the feasibility of introducing orthogonal persistence into the C++ programming language with minimal disruption to the language specification. It eventually evolved to include another object-oriented language, namely Java. The approach that was chosen was to provide orthogonal persistence to the language by way of a run-time environment inside which compiled programs execute. This is the well-known and proven Napier88 approach that has been operational throughout the many releases of the Napier88 language environment.

Models of distributed persistent programming systems were also examined. In this thesis, the approach taken with respect to distribution has been that a confederated model is the most practical architecture given the reality of failure, organisational barriers and geographically distant sites. Only within a single organisation where security can be relaxed and store independence compromised, would the one-world distribution model be workable. Diagrammatically, a confederated architecture advocates a model of application building similar to the FIDE approach of Figure 1-1, except for a distributed repository of data and operations. This is shown in Figure 9-1 overleaf.

Finally, an initial implementation of the proposed distributed architecture has been developed. Issues about performance have also been discussed. The difficulty of identifying objective performance indices and interpreting the measurements obtained was also highlighted.

9.1. Thesis Contribution

The contribution of this thesis work is that it has:

1. ascertained that orthogonal persistence can be supported in a mixed-mode language like C++,
2. developed and implemented a model of confederated persistent object stores suitable for wide area networking,
3. developed and implemented a containment mechanism for structuring a large, potentially distributed persistent space, and,
4. shown that parameter passing by indirection is suitable in situations where by-reference semantics are needed without the side-effects of using direct references.

The above goals were supported by both quantitative and qualitative assessment of implementations performed as part of the work described in chapter 6.

135
The mechanisms discussed in this thesis can be used to either build confederated store arrangements or as a “glue” to support interaction between single store images. The distribution environment is one where stores are largely self-sufficient, with an occasional need to access a remote store for additional resources. The model targets wide area networks and it takes into account autonomy and failure as primary concerns.

9.2. Future Directions

A number of enhancements can be made to the indirection mechanism proposed in chapter 6. In particular, in its current form, there is no support for asynchronous remote operations. A full-blown implementation could also provide for a non-blocking version of Indirection.invoke so that clients need not wait for the remote server to send back results. In that way, a client application would go on with its local computation whilst the server store processes the requested operation. For example, say asynchronous operations are requested through a call to the method request_operation of class Indirection which returns a capability to the client. When the server finally sends the results, it does so to the client’s store who holds on to them in a transit area. Eventually when the client application is ready to inspect the results, it requests them from its home store. So the client could execute statements such as those shown in Figure 9-2.
9. Conclusion

// Indirect to an image processor service.
Indirection improc = new Indirection ( "image.processor.edu.au", "the_image_processor" );
// Build an array of arguments for remote method detect_edge.
Object args = [ new PGM_image ( "an_image.pgm" ), new Integer ( 150 ) ];
// Make the asynchronous request and obtain a capability.
Capability ticket = improc.request_operation ( "detect_edge", args );

// Here, this application goes on until it is ready to check results of “detect_edge”.
// So, do some more work here... and finally, find out if results are ready.
PGM_image edges = ( PGM_image ) Store.get_results ( ticket );

Figure 9-2: Possible asynchronous service request.

With respect to localities, the proposal of the concept is only a first step towards self-contained partitioning of a potentially distributed persistent space. More work is required to ascertain the suitability of this approach to specific application areas. The image processing application used as example in chapters 7 and 8 was chosen because its components were easily identifiable and it was easy to structure the application in a distributed fashion. Further investigation into the following would be desirable:

1. The kinds of applications that would benefit from localities.
2. How to control component failure through localisation.

Once these two are ascertained, several research areas would benefit from the use of localities and indirections. A few of them are briefly introduced in the following subsections.

9.2.1. Distributed Hyper-Programming Environments

Whilst we proposed the indirection and locality concepts as tools that enable the development of distributed hyper-programming systems, it is exactly what we have: a proposal. However convincing preliminary results may be, showing that these concepts do work is really a far cry from developing a fully operational distributed hyper-programming system. Therefore, given more research effort and financial commitment, this kind of development would be a worthy undertaking. One could envision such an endeavour resulting in a product like the Napier88 Programming Language Environment adapted to a distributed environment. Such a product would incorporate distributed browsing so that a user can inspect the contents of remote stores by querying them through an indirection.

9.2.2. Component Architecture/Frameworks

A framework is a re-useable, “semi-complete” application that can be incorporated into a wider computing environment to produce custom applications [Johnson & Foote 88]. In contrast to existing code re-use techniques, e.g. libraries, where code copying is practiced, frameworks support true sharing, well-defined interfaces and modularity. However, the majority of framework implementations are of the kind that could be called “white-box” frameworks in that the relationships among them are static and they are sometimes not opaque to application developers. JavaBeans [Sun 99c] are an example of such frameworks and are especially meant for use in a distributed environment. With the increasing pervasiveness of the World-Wide Web, network-aware approaches to aggregating and delegating responsibility are assuming increasing importance in distributed computing. Products such as Sun’s Beans Development Kit [Sun 99c]
provide a way to create components targeted for distributed environments. Given a Bean the distributed application programmer is then able to inspect the content of the component and using specific interfaces provided, make use of the Bean resources.

Recent literature on frameworks [ACM 97] predicts that a different flavour of frameworks, of the so-called “black-box” kind will be a future trend. Black-box frameworks emphasise dynamic relationships among components as opposed to the static class relationships. Therefore they are expected to be easier to dynamically reconfigure or extend.

In view of the above, it does appear that localities are a suitable containment mechanism for encapsulating black-box components. In terms of the properties exhibited, a locality, with its embedded service is no different from a framework. We can go further and envisage that the owner of a black-box could even deploy the black-box as a stealth object on the network. Thereafter, copies of the indirection to the stealth would be distributed to interested clients. The clients then call back without even having to know the physical location of the black-box.

9.2.3. Localities and Indirections in Metacomputing

Localities and Indirections could also be used in building metacomputers—a metacomputer being a network of autonomous, possibly heterogeneous computing resources linked by software [Smarr & Catlett 92]. The integration of an ever-changing pool of resources constitutes one of the most difficult issues in building metacomputing systems. This is due to the fact that scalability becomes an important factor and because such systems are intended to be massive, nothing short of absolute scalability will suffice.

In chapter 5 of this thesis, the very first assumption made in developing the confederated model was that of “true distribution”. In other words, no node in the distribution depends on any centralised resource for its operation. We believe that this assumption is, or ought to be of prime importance in the development of metacomputing environments. With this assumption in mind, resources would then be identified and organised into autonomous localities and finally applications written to exploit the localised resources through the use of indirections.

Communication software is a major component that underlies any distributed environment. A first stage in the development of a metacomputing environment is the provision of a uniform mechanism to abstract over hardware differences. In this respect, it is likely that indirections can prove to be useful in achieving such abstraction.

9.3. Finale

For economic reasons, workstations and servers built from commodity processing and networking components are becoming increasingly popular. Additionally, advances in wide-area networking and the wider availability of this technology has meant that organisations of modest means are able to afford such networking technology. Therefore it is foreseeable that the distributed high-performance computing environment of the future will be loosely-coupled computers connected to high-latency, low-reliability inter-connects with little security guarantees. Such an environment advocates strongly towards the model proposed in this thesis, and models like this one will play increasingly important roles in the future of large-scale persistent stores.

It is a fact that wide area networking technology is rapidly improving so that eventually, WANs are expected to equal LANs in terms of reliability and bandwidth. In practice though, despite
technological break-throughs in WAN technology, organisational barriers are bound to make one-world implementations difficult to instantiate.

Inherent in the confederated store model, is the respect for organisational barriers. In this model, the stores that make up the virtual global space may well belong to separate organisations, each with its own confidentiality and control policy. Such barriers are real-life impediments that arise outside the computation realm; they cannot be modelled using any software abstraction and they may not be ignored. Ignoring them allows the formulation of one-world models that ultimately exhibit the inability to scale beyond the barrier. Therefore, throughout this thesis, the stance advocated has been that organisational barriers are to be reckoned with, hence the adoption of Munro’s confederated model as the basis for the work described here.


CERN (1990). Where the Web was born, CERN European Laboratory for Particle Physics, http://www.cern.ch/Public, Switzerland.


