SCHEDULING IN METACOMPUTING SYSTEMS

By
Heath A. James, B.Sc.(Ma. & Comp. Sc.)(Hons)
July 1999

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ADELAIDE

Contents

List of Tables

List of Figures

Abstract

The problem of scheduling in a distributed heterogeneous environment is that of deciding where to place programs, and when to start execution of the programs. The general problem of scheduling is investigated, with focus on jobs consisting of both independent and dependent programs. We consider program execution within the context of metacomputing environments, and the benefits of being able to make predictions on the performance of programs. Using the constraint of restricted placement of programs, we present a scheduling system that produces heuristically good execution schedules in the absence of complete global system state information. The scheduling system is reliant on a processor-independent global naming strategy and a single-assignment restriction for data.

Cluster computing is an abstraction that treats a collection of interconnected parallel or distributed computers as a single resource. This abstraction is commonly used to refer to the scope of resource managers, most often in the context of queueing systems. To express greater complexity in a cluster computing environment, and the programs that are run on the environment, the term *metacomputing* [74] is now being widely adopted. This may be defined as a collection of possibly heterogeneous computational nodes which have the appearance of a single, virtual computer for the purposes of resource management and remote execution.

We review current technologies for cluster computing and metacomputing, with focus on their resource management and scheduling capabilities. We critically review these technologies against our Distributed Information Systems Control World (DISCWorld).

We develop novel mechanisms that enable and support scheduling and program placement in the DISCWorld prototype. We also discuss a mechanism by which a high-level job's internal structure can be represented, and processing requests controlled. We formulate this using extended markup language (XML).

To enable processing requests, which consist of a directed acyclic graph of programs, we develop a mechanism for their composition and scheduling placement. This rich data pointer and a complimentary futures mechanism are implemented as part of the DISCWorld remote access mechanism (DRAM). They also form the basis for a model of wide-area computation independent of DISCWorld.

We have implemented geospatial imagery applications using both simple RMI and common gateway interface, and the novel mechanisms developed in this thesis. After analysing and measuring our system, we find significant performance and scalability benefits.

Declaration

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 Library, being available for loan and photocopying.

Although DISCWorld is a joint development of the Distributed & High Performance Computing Project group and other students, the work reported here on scheduling in DISCWorld is my own work.

Heath A. James, B.Sc.(Ma. & Comp. Sc.)(Hons)
July 7, 1999

Figure 1: Calvin and Hobbes ©Watterson. Reprinted with permission of Universal Press Syndicate. All rights reserved.

Acknowledgements

There are of course a great many people to whom I am indebted in various ways for their support during the years that I have been engrossed in the work reported herein.

First and fore-most I would like to thank my wife and best friend, Christie, for *everything*. For everything including, but not limited to: encouraging me when I was down; listening to me when I was trying to work through a problem; leaving me alone when I needed it; making me work when I didn't want to but should have; for making me take breaks; for simply everything.

I am indebted to Mum and Dad, for their love and support during my candidature. For letting us come over and simply be ourselves, without the need for any ceremony. Thanks, too, Dad for being my house-hold fixer while I've been too busy to take care of the odd-jobs around the house.

To Sue and Bruce I extend my warmest gratitude for the continual love, support and encouragement that you have shown to me (and us) while I have so often been distracted by thoughts on my work. You, too, have been a pillar of support.

My sincerest thanks to my supervisor, Ken Hawick, for all his encouragement, patience and friendship over the past 3 years. The long conversations over coffee and cake will always be remembered fondly.

To my fellow members of the Distributed & High Performance Computing group: Ken, Paul, Francis, Kevin, Andrew, Duncan, Katrina, Din, Craig, James and Jesudas. Thanks for the good times. I have really enjoyed working with you all – and I hope to continue doing so in the future.

Thanks to Kim Mason for all his Java Advanced Imaging help. I surely wouldn't have solved some of the implementation problems without the benefit of his experiences.

To all those people with whom I have shared offices: Dean, Matt, Ken, Paul, James, Jesudas and Duncan. I realize working next to me can be somewhat interesting

at times – I just hope I've inflicted a small piece of insanity on you all.

Yay ... I finished!

This work was carried out under the Distributed High Performance Computing Infrastructure Project (DHPC-I) of the On-Line Data Archives Program (OLDA) of the Advanced Computational Systems (ACSys) Cooperative Research Centre (CRC) and funded by the Research Data Networks (RDN) CRC. ACSys and RDN are funded by the Australian Commonwealth Government CRC Program.

I would also like to thank Bob Gingold and Judy Henderson, from the Australian National University Supercomputer Facility, for supplying me with performance data from the Silicon Graphics Power Challenge and the Fujitsu VPP300.

Chapter 1

Introduction

Metacomputing [74, 211] is an abstraction by which clusters of computers in a distributed system can be treated as a single virtual computing resource. Scheduling is an important aspect of distributed computing, particularly in the case of metacomputing systems, where the scale of the system, and resource heterogeneity give rise to a large number of scheduling choices.

We have reviewed the current technologies in metacomputing systems and specifically the way in which scheduling, or program placement across distributed systems is approached. We have concluded that with few exceptions, scheduling is performed in an *ad hoc* manner. Therefore, there exists an opportunity to study the way in which schedules for distributed systems are created and executed, and to apply this research to metacomputing systems.

We consider processing requests that can be decomposed into relatively coarse-grained computations interconnected by data dependencies. We explicitly include the case in which it may be more economical to move the computation to where the data resides, rather than to move the data to the computation.

We initially consider the effects of a number of scheduling algorithms on the total execution time of independent programs with execution times chosen from a given distribution. We conclude that no single scheduling algorithm performs optimally with all distributions of execution time across both homogeneous and heterogeneous distributed clusters. We suggest the use of an adaptive scheduling algorithm which initially creates a static execution schedule that is adapted by knowledge of partial system state to minimise execution time.

The task of maintaining complete, accurate system state information in a distributed computational environment is very difficult. We hypothesise that an

adaptive scheduling algorithm should be used to enable the performance of a metacomputing system to be optimised from two differing points of view: that of the user, in turnaround time; and, that of the system, in the impact that computations have on the system in the long-term. The scheduling algorithm makes use of an individual node's knowledge of the remainder of the global system state to generate optimal schedules, according to a given heuristic.

We present a set of abstractions with which we can describe the characteristics of data and programs, and the assumptions which make them valid in a general distributed system. These abstractions are used to create adaptive execution schedules with which a processing request may be satisfied and optimised. The experimental framework is implemented as an integral part of the broader Distributed Information Systems Control World (DISCWorld) metacomputing system.

This thesis does not attempt to provide solutions to all the research problems faced when implementing a metacomputing system. The focus of this work is on *scheduling*; we develop an algorithm and mechanisms for enabling smart scheduling of programs in the DISCWorld metacomputing system.

1.1 Project Motivation

In the nearly forty years in which computers have been available, the user base has broadened from isolated groups of mathematicians and engineers to the point where they have become widely accessible and affordable commodity items with applications for scientific research, business management and entertainment. Their ubiquity is driving users from many disciplines to use computers.

For those users with large high-performance computing requirements, the trend has been to move away from expensive massively parallel machines to medium- or small-scale, general-purpose parallel machines and clusters of workstations, as well as combinations of distributed heterogeneous machines [6]. Together with the widespread adoption of computers into daily life, computers are also being used to gather and store data on a wide variety of phenomena [133, 167]. New equipment is being deployed which can produce many terabytes of data per day, such as multi-spectral Earth observation satellites [17]. With a proliferation of data being collected every day on various physical phenomena (including visual spectra, invisible spectra, geothermal, seismic, hydrology), people are using this data to produce new, *value added* data.

The application domains of environmental science and decision support are areas that require access to a wide variety of data that is characterised by being very large and expensive to produce. Common software packages that are available for use by environmental scientists [59] are unable to handle the volume of data required for processing. Thus, more users are turning to high performance computers, and collections of computers, to provide the necessary processing capabilities. Unfortunately, these people are not often computer science specialists, and may not possess the skills to effectively program and use high-performance computing hardware that may be beneficial to them.

One of the characteristics of these application domains is the fact that, as the data is very large, it is also expensive to transfer across a network. Data is also being collected by different organisations [17, 35, 133, 164, 225] by different methods (including satellite, ground observations, radar data) and each owner may stipulate different conditions on the use of their data. The data may also be stored in repositories that reside in different parts of the same country, or even in different countries. Decision support systems often require near real-time results, so performance is a major consideration. The issues involved in implementing a distributed system that provides a computational infrastructure for the development of decision support and research applications are presented in appendix A.

The problem of enabling non-computer scientist users access to large-scale distributed systems is not trivial. Technical issues such as ensuring each machine has a standardised interface which hides the heterogeneity of the platform and interconnection networks must be addressed. In summary, what these users require is an abstraction over the computers and data repositories that make up their computational infrastructure, which does not require them to be computer science experts.

A prototype system which provides this functionality is described in appendix B. The Earth Observation Information Catalogue (ERIC) system allows authorised parties access to satellite imagery stored in an online data archive, and permits simple operations on the images and metadata using basic web-based client-server technologies. The inadequacies in the system have contributed to the design and philosophy behind this project.

1.2 Why Metacomputing?

With the advent of networking technologies such as Ethernet [50] and ATM [103] it has become possible to connect computers for the widespread, efficient sharing of data [221]. As high performance local- and wide-area networks have become less expensive, and as the price of commodity computers has dropped, it is now possible to connect a number of relatively cheap computers with a high-speed interconnect, to effect a local distributed computing cluster [115].

The granularity of code that is being run across collections of computers is becoming greater, and the need for frequent synchronisation is decreasing. For example, in massively parallel computers, such as the Thinking Machines CM-2 and the CM-5 running in single-instruction, multiple data (SIMD) mode, processors synchronously execute instructions, albeit on different data. This is feasible due to the high bandwidth and low latency of communications between processors. In local-area clusters of computers, where the interprocessor bandwidth may be comparable to that found in a supercomputer but latency is higher, stand-alone tasks that operate on a fraction of the total data to be processed by a program may be run on each processor (MIMD). Thus, while the cluster of computers and the SIMD supercomputer may have the same amount of data to process, the application executing on the cluster will probably perform less synchronisation between processors. Furthermore, the trend is to move towards service-based computing, where stand-alone programs, requiring very little, if any, synchronisation, are run on distributed computers, which may themselves be massively-parallel processors (task parallelism). Hence, variable interprocessor bandwidth and latency characteristics are tolerable.

Clusters of distributed computers, as well as high performance serial and parallel machines can be interconnected, speaking common communications protocols, to form a large virtual supercomputer [2, 39, 69, 95, 113]. The general trend for computational capacity has thus been to move from monolithic single-processor computers in the early 1970's, to multi-processor parallel computers, in which the interprocessor bandwidth was high and the latency quite low, to clusters of commodity workstations with comparatively high bandwidth and latency, and ultimately to so-called metacomputing environments, which connect heterogeneous clusters of high-end and low-end computers to form a virtual supercomputer.

The term *metacomputer* [74, 211] was coined to describe a collection of possibly heterogeneous computational nodes which can be treated as a single virtual computer for both resource management and remote execution purposes [19]. General

metacomputing environments allow users to submit serial or parallel programs and have tasks or jobs run on the virtual computer. An alternative definition of metacomputing is provided by Gehring and Reinefeld [78]: "a monolithic computational resource provided by software that allows the transparent use of a network of heterogeneous, distributed computers".

1.3 Focus and Contributions of this Thesis

Users are attempting to solve larger problems, using data which may not be stored at one physical location, or are not able to be contained within a single cluster of computers or single multi-processor. This provides incentives for users to join metacomputing environments, and owners to join larger clusters of computers, providing a higher aggregate compute capacity for users, which increases the chances of higher resource utilisation. Of course, by allowing users access to other clusters of computers, owners open their own resources to use by the wider community in a *quid quo pro* relationship.

We have examined a number of different systems [2,39,69,95,153,194] that provide approaches to metacomputing, but we believe that each of these systems suffers from deficiencies which make them inappropriate for the problem domain of high-level decision support services. For example, some systems focus more on the co-allocation of processors [69], while others focus on the effective use of idle machines [153]. Still others focus on the compositional nature of collaborative systems [39]. We address these deficiencies by introducing an environment model in which users can submit high-level processing requests to a daemon which runs on a local machine. The request is parsed by the daemon and decomposed into smaller, self-contained programs which together can access and process the data to fulfil the user's request. From the user's point of view, the system consists of a single virtual machine to which they submit requests, and can retrieve results.

One of the most important aspects of a metacomputing system is the method by which programs are allocated to processors in a fashion that will yield the best results from the viewpoint of both the user and of the owners of the resources. This problem is called the scheduling and resource management problem, and it is this man focus of this thesis.

El-Rewini [54] and Ahmad [6], amongst others, recognise the need for developing novel techniques for the management of computing resources through highly efficient

dynamic task allocation, scheduling and load balancing algorithms. One of the major problems that the wider community studying scheduling systems has not addressed is that of not being able to place tasks onto every node in the distributed system. While this limitation has been alluded to by the introduction of restrictions based on machine (and hence binary code) heterogeneity, we wish to address the problem of controlling the system from the viewpoint of a platform independent language, such as the Java programming language, in which tasks, or *services*, are written and are distributed throughout the machines in the system. Some of the services in the system may be implemented as native programs, encapsulated in Java wrappers. Although services implemented in pure Java are physically able to execute on any of the nodes (due to the platform independence of the object code or byte code), they can be restricted in their choice of machine to be run on by virtue of machine-dependent *policies*. Policies are controlled by the machine's owner or administrator. This has an effect on the manner in which the data and programs in the system can be used and transferred, and also in the way that the scheduling system must adapt to the conditions.

1.4 Organisation of this Thesis

In order to properly place metacomputing into its historical perspective, a review of current and influential cluster computing projects is presented in chapter 2. The classical approaches of cluster control via batch queueing systems and wide-area managers are discussed. The concept of metacomputing, and the characteristics of metacomputing systems are discussed. The chapter concludes with an introduction and discussion of the Distributed Information Systems Control World (DISCWorld) metacomputing project, in which this scheduling work is developed. We critically compare DISCWorld to the other systems reviewed in chapter 2.

A literature survey on scheduling research, with focus on processor selection and allocation, is presented in chapter 3. The conclusion from this work is that there is very little scheduling research that considers the problem of restricted placement of code, and that considers both code and data to be movable between nodes in a distributed system.

Chapter 4 presents work that was performed in the area of scheduling independent tasks across a cluster of distributed computational resources. Simulations were performed using a number of different static and dynamic scheduling algorithms under

conditions that restrict the use of the systems, as mentioned in chapter 2. We reach two important conclusions: firstly, that there is no single scheduling algorithm that produces the best execution schedule when there are greatly varying numbers of jobs to be executed, with different job execution time distributions; and secondly, that the classifications of scheduling algorithms into either static or dynamic is too restrictive. We suggest that new descriptive terms, representing hybrid approaches in the presence of complete or partial system state information, are needed.

Scheduling, in the presence of restricted placement and movable code and data is the focus of chapter 5. Scheduling is modeled in accordance with the features and restrictions found in the prototype DISCWorld metacomputing system. The resulting model is general to distributed systems, even in the presence of partial system state information. Partial system state information is evidenced in typical distributed environments, where elements of the system are perhaps owned by different organisations and access by the metacomputing is subject to the owner's policies. Our model requires that there exist a global naming scheme for data and code, and single assignment of data. In this context, single assignment of data implies that once data is created and given a canonical name, any modification of the data will result in new data being created, having a different although related name. A description and discussion of the implementation of the scheduling model is presented. Also featured is a brief discussion of work on a distributed job placement language (DJPL). User requests are represented by process networks of high-level programs which share data in a producer-consumer relationship. The purpose of the DJPL is twofold. Firstly, it is used to express the user query when decomposed into programs and shared data; secondly, it also describes the static assignment of programs to machines within the DISCWorld system. In addition, the DJPL contains information that describes the user request and the parameters under which it may run, and also details the appropriate behaviour of the system when an exception condition is raised. We believe that the DJPL is sufficiently general that it may be effectively integrated with other high-level metacomputing systems.

Execution mechanisms associated with the scheduling model are presented in chapter 6. We describe a novel data access mechanism, the DISCWorld Remote Access Mechanism (DRAM), and discuss the manner in which it is used in our implementation of the DISCWorld model. The DRAM concept is extended to refer to data which has not yet been created, the Future DRAM (DRAMF). The DRAMF facilitates the execution of schedules in the DISCWorld metacomputing environment

and allows executing schedules to be dynamically optimised to achieve the best performance using available state information. This optimisation is evidenced by possibly faster request execution time for the user and a lower overall system impact.

Performance analysis of the DISCWorld prototype is presented in chapter 7. The system is compared with ERIC, a simple client-server system of the same functionality, and an RMI implementation of ERIC. Limitations of the implementation are discussed.

The thesis is concluded, and future work is described in chapter 8. Other relevant work is summarised and presented in appendices A and B. Appendix A is a summary of a paper detailing the motivation for the concept behind the DISCWorld system philosophy. Appendix B describes a user interface to a satellite data repository that was built in the beginning of this work. It describes the way in which a simple client-server technology, such as CGI [100] can be effectively used to provide server-side computation across the WWW. We show that the technology is inadequate for the types of systems detailed in appendix A due to: its speed; the total reliance on server-side computation; the poor client support; and the fact that the technology is not scalable to complex distributed systems.

Chapter 2

Cluster Computing

This chapter provides a review of the software that has been developed in response to the trends mentioned in section 1.2. The remainder of this chapter is further divided into four sections: a summary of resource management technologies; a description of metacomputing projects; an introduction to our prototype metacomputing project, Distributed Information Systems Control World (DISCWorld); and a discussion in which other systems are compared and contrasted with DISCWorld.

The evolution of computer systems from single monolithic mainframes, to massively parallel machines, to groups of tightly (and loosely) interconnected distributed workstations, has forced a change in the manner in which the systems are used, and what can be achieved with the systems. As computers, and groups of computers, become more complex, greater abstractions are necessary so ensure that users are able to exploit the computational power effectively, and to ensure that the resources are fairly and efficiently used.

While there is much debate about exactly what a cluster *is* [120], in this thesis we use the definition proposed in [186]: a *cluster* is a collection of interconnected parallel or distributed machines that may be viewed and used as a single, unified computing resource. Clusters can consist of homogeneous or heterogeneous collections of von Neumann (serial) and parallel architecture computers, or even sub-clusters. An example of a cluster is shown in figure 2, where servers $1 \ldots n-1$ are workstations and server n is an interface to a sub-cluster. The term cluster usually refers to a tightly connected group of computers, connected by a high-speed interconnection network.

A resource manager is the name given to the software that provides management functions and abstractions that allow clusters to be treated as a single resource. Furthermore, resource management software is employed by system managers to

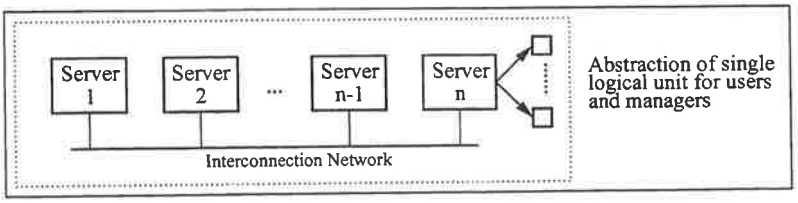


Figure 2: A cluster of computers, where servers $1 \ldots n-1$ are workstations and server n is either an interface to a sub-cluster or a multi-processor machine. Any machine may provide computational and cluster-management functions.

ensure resources are available depending on a pre-defined criteria. Resource management software can be used, for example, to ensures that all users' jobs are executed in turn and hence receive the same amount of system resource; or, by not allowing all users to immediately execute their jobs, load on the machine is spread out and the shared resource can be effectively utilised.

There are a number of specialised resource management software products available. These may divided into batch queueing systems and extended batch systems. Batch queueing systems are designed for use on tightly interconnected clusters, which usually feature shared file systems. Extended batch systems, designed for use in loosely interconnected clusters, do not usually make assumptions about shared file systems, and often offer increased functionality over typical batch queueing systems. Examples of batch queueing systems are: DQS [91, 220]; GNQS [139]; PBS [116]; EASY [151]; LSF [188]; and LoadLeveler [123], while examples of extended batch systems are: Condor [153]; PRM [172]; CCS [195]; and Codine [81].

2.1 Resource Managers

Baker, Fox and Yau [20, 21] divided cluster computing software into two groups: cluster management software (CMS); and distributed cluster computing environments (DCCE). Their definition of CMS maps onto our classifications of batch queueing systems and extended batch systems. Cluster management software systems are typically characterised by the user submitting a job description file specifying the programs to be run and the types of resources that are necessary for them to run. The resource management system selects an appropriate host on which to run the job, and plays no further part in the execution until either the job completes successfully, or needs to be killed for consuming too much resource. Finally, the resource manager places the outputs of the program execution at a place nominated by the user for

later collection. There are various forms of result notification, including the manager placing the results in the directory from which the batch job was submitted or the user being sent email with the location of the results. Thus, batch queueing systems do not individually schedule the programs that comprise a job; they merely place the job onto a sufficiently unloaded server. The only form of scheduling they perform is the matching of available computational capacity with the job's requirements. The user typically submits their job to a queue that best reflects the job's characteristics (e.g. short, medium, or long execution time; priority based; short, medium, or long job execution time requiring parallel processors). In addition, batch queueing systems have no real support for platform heterogeneity. Users must ensure that binaries are available for the architecture on which the job will be placed.

An idealised batch queueing system is shown in figure 3, where jobs are submitted to a centralised queue master, placed by the queue master onto a compute server, and the output is returned to the user after processing is complete. Users typically have no control over which server executes their job. In contrast, an extended batch system is shown in figure 4. Users can submit their job to any server participating in the cluster. If the server to which the job is submitted does not have the available computational capacity to handle the job, it is moved to another server.

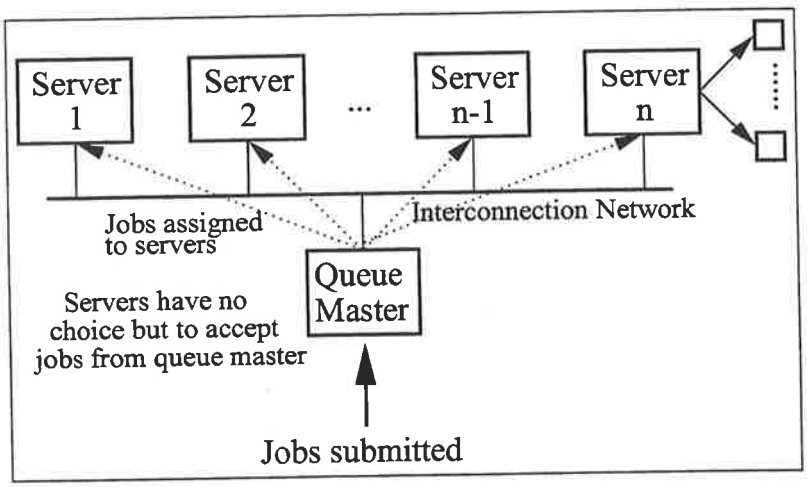


Figure 3: Architecture of a typical batch queueing system. Jobs are submitted to the queue master. The queue master assigns jobs to a server. When the job has finished executing, the results are returned to the user.

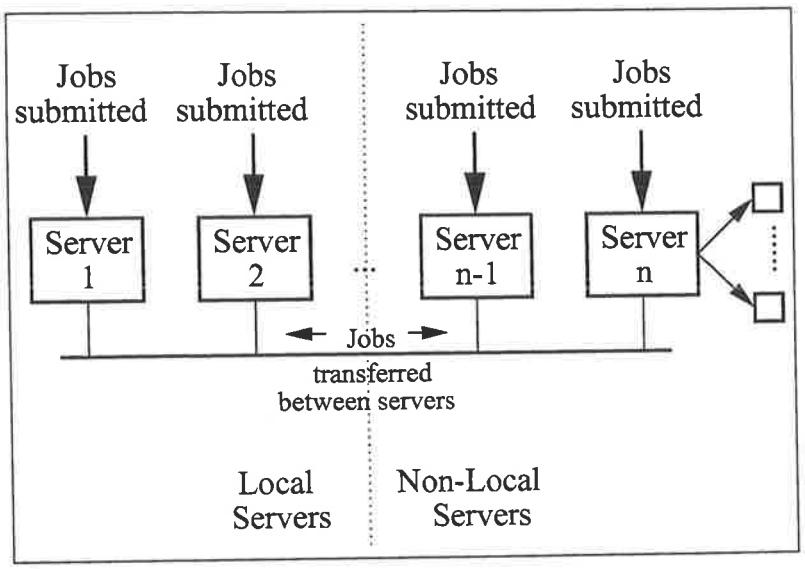


Figure 4: A wide-area manager. Jobs may be submitted to any server. If the local server is unable to execute the submitted job, the server decides where to send the job. Some systems use limits on the maximum number of times a job can be transferred to ensure it is eventually executed. The scale of network is larger and servers can be more loosely interconnected than in a batch queueing system.

2.1.1 Batch Systems

The two main reasons for using a batch system are: to maximise the use of shared resources; and, to make sure everyone can effectively and equitably utilise that resource. They are not predominantly used to utilise spare cycles on users' machines, although many packages provide this functionality [153].

As the name suggests, batch systems are only useful for the initiation and controlling of batch jobs. Although it is common to submit jobs to a batch system as a shell script, queue masters do not interpret the instructions contained within the script. Most systems merely parse the script for job-identification tokens, such as the name of the user that submitted the job and what resource usage is expected by the job. The script is then executed on the server to which the job was assigned, which runs the instructions contained therein. Thus, batch queueing systems are of little use when presented with a job that consists of a series of dependent programs which may be interactive, and may be possible to execute in parallel. These are the type of jobs often submitted to a metacomputing environment.

While the batch queueing systems described in this section are used successfully

with clusters of workstations, one of the conclusions of Baker *et al* [21] was that software being developed for compute cluster environments and distributed compute cluster environments provide the most viable means of utilising workstation clusters. The main reason for this is that compute cluster environments and distributed compute cluster environments provide more flexibility than accessing machines individually. Batch queueing systems often assume that they are the only schedulers executing on the resources.

Through experience, we have discovered that GNQS, DQS and PBS are fragile when the machine that runs the queue master has multiple network interfaces. If the queue-master machine is used as a front-end to a cluster of machines that can only communicate through the front-end, these batch systems fail. We believe that this is due to the batch queueing system only recognising the front-end machine's primary network interface. Secondary interfaces, such as those that connect to the cluster, are ignored. This problem can be solved by using a wide-area resource manager designed for such conditions, such as Condor.

In a distributed computing context, with the aims of providing infrastructure for decision support systems, batch queueing systems are not entirely useful. While they provide effective resource utilisation, they are not able to handle requests which are comprised of dependent programs, especially if the programs can be logically run in parallel. In addition, they do not support the wide-area model in which the data to be used by a program is not immediately accessible.

GNQS

The development of Network Queueing System (GNQS) was driven by the need for a good UNIX based batch and device queueing facility capable of supporting such requests from a networked environment of UNIX machines [117, 139]. Later superseded by Generic NQS (GNQS), it was implemented as a common interface between machines, using a collection of user-space programs to provide both the batch and device queueing capabilities for each machine in the network. GNQS relies on the cluster of workstations sharing a common file-system and common user database.

Within GNQS, a batch request is defined as a shell-script containing commands, that can be executed without any user intervention by an appropriate command interpreter. The commands contained within a batch request cannot require any specific peripherals. A device request, on the other hand, is defined as a set of instructions requiring the direct services of a specific device for execution.

GNQS has support for all the resource quotas that can be enforced by the underlying UNIX kernel implementation, and can support remote queueing and routing of batch and queue requests throughout the network of machines running GNQS. GNQS also supports queue access restrictions, and allows user accounts to be mapped across machine boundaries. Command standard output and error can be returned to the user on the originating (possibly remote) machine. Remote status operations are supported to alleviate the need for remote users to log into a machine in order to obtain information. The designers of GNQS also provided for the inclusion of file staging, which was to be implemented in a future release; unfortunately this has not happened.

PBS

Portable Batch System (PBS) [23, 116] is a package which was designed and written by the Numerical Aerodynamic Simulation Complex, NASA, from 1994. PBS is a successor to NQS, and addressed many of the deficiencies of NQS. It was designed to provide additional controls over the initiating, or scheduling, of execution of batch jobs.

PBS extends the UNIX operating system by the addition of user-level services. Unlike NQS, it had detailed design and implementation documentation for developers. It was designed to be easy to add functionality and improved over NQS in a number of ways, including the provision for parallel jobs. PBS also included features that allow the scheduling policy to be modified according to a site's needs. A batch scheduling language has been designed for use with PBS, but the documentation recommends that it not be used, as there are implementation problems.

DQS

Distributed Queueing System (DQS) [91, 220] is a batch queueing system from the Supercomputer Computations Research Institute, Florida State University, which uses shared files to manage user-submitted jobs. It uses limited system information (the instantaneous and past load on a machine) and has no real support for binary heterogeneity. Managers can set up queue complexes, but the implementation is fragile, and must be monitored. It is fault tolerant, able to restart jobs on machines that have failed. In order to use the system, users must be familiar with the creation of shell-scripts in which they place the program(s) to be run. DQS supports parallelism via the PVM [79] system.

LSF

Load Sharing Facility (LSF) [187, 188], by Platform Computing, is a very popular commercial batch queueing system. Like most batch queueing systems, LSF relies on the existence of shared files to implement queues, locks and logs. In addition to standard features found in batch queueing systems, LSF has some measure of fault tolerance inbuilt. In the event a shared file system is not available, the degree of fault tolerance is reduced. If the master host (which makes scheduling decisions) fails, another host in the cluster is automatically voted to be the master. Hosts are elected to be the master in the order they appear in a static file which must be visible to all machines in the cluster. In the event that the cluster becomes partitioned by network failure, the partition that has access to the LSF log files continues working, while the remaining partitions sit idle.

Unlike GNQS and PBS, LSF supports machines with multiple logical names. Jobs are scheduled according to the availability of available hosts, their resource requirements, and whether the job should be able to finish before the queue ceases to run. LSF provides job check-pointing and migration facilities for some machines [188]. It is possible to express dependencies between complete jobs so that dependent jobs are only run once their dependencies have been completed. Because LSF cannot be expected to run on all nodes in a distributed cluster, it has the ability to submit jobs to an NQS batch queue.

LSF has the ability to run parallel jobs through the use of PVM [79]. The user must submit their job using a non-standard command (`pvmjob`). When the job is executed, LSF creates PVM daemons on each host in the cluster and then starts the user's job.

LoadLeveler

LoadLeveler [123], by IBM, is a modified version of the Condor [153] batch queueing system. LoadLeveler supports clusters of workstations and multi-processor machines. It has parallel support for batch and interactive jobs, and is compatible with NQS. LoadLeveler has an application programmer interface (API) which allows TCP/IP applications to directly connect with least-loaded cluster resources.

The Extensible Argonne Scheduling sYstem (EASY) [150] was incorporated into LoadLeveler to produce EASY-LL [149, 210]. EASY provided a better scheduling algorithm through which jobs could be selected to run. Jobs are ordered in the submission queue by their submission time. Jobs are considered for execution in

this order. If the scheduling system has enough free nodes to run the first job, it is run. Otherwise, the first job is assigned the start time of the currently-running job's finish time. Jobs requiring fewer resources are executed as long as they will complete before the first job is due to start. Thus, jobs are back-filled to make better use of machine resources. The EASY-LL scheduling system was one of the first to provide deterministic, yet opportunistic scheduling algorithm based on the bin-packing algorithm [76].

2.1.2 Extended Batch Systems

Extended batch systems are characterised by having additional functionality over batch queueing systems. For example, some systems have the ability to act as disk caches [12], to harness idle CPU cycles on participating machines while the owner is not using them [12, 153] or to manage software licenses [81]. Still others make searching nodes in the cluster easier for fine-grained parallel tasks [172]. The remainder of this sub-section provides a description of four significant extended batch systems: Prospero Resource Manager, Codine, Condor and Computer Center Software.

Prospero Resource Manager

The Prospero Resource Manager [171,172] (PRM) is designed for use with distributed and parallel programs implemented in PVM [79]. It presents a uniform and scalable model for scheduling tasks by providing mechanisms through which nodes on multiprocessors can be allocated to jobs.

The main contribution of PRM is its hierarchical organisation of system resources into a framework that can be easily searched for use by fine-grained parallel applications. Architecturally, PRM is divided into two distinct parts: the system and jobs. The system part is concerned with the resources that comprise the system, which are global and local. When job execution is requested by the user, the tasks that comprise a job are allocated to different resources. System management resources are allocated to jobs as they are needed, and resources are managed separately for each job. Multiple resource managers are used to achieve resource management scalability, each controlling a subset of the resources. There are three types of managers: system managers, job managers and node managers, each of which uses a different level of information abstraction.

The system manager controls a collection of physical resources, allocating them to jobs when requested. It collects and manages all the information pertaining to the resources under its control. The job manager requests resources needed by a particular job, and assigns them to individual tasks in the job. The job manager's only concern is the job to which it is allocated. The node manager loads and executes tasks on a local machine, as requested by the job manager and authorised by the system manager. The managers organise information using the Prospero Directory Service [169,170], which is based on the Virtual System Model. System managers are organised in a hierarchical fashion.

On job execution, decisions on which resources to use are determined by the job manager querying the directory service for a suitable resource set. During execution, the job manager monitor the tasks, and passes any requests for additional resources to system managers. Resource heterogeneity is handled by the use of architecture identifiers in the directory service.

Unfortunately, the only way in which Prospero can execute parallel job is by the use of the PVM system. This introduces restrictions on the programs that can be used with the system. In the context of decision support infrastructure, we do not wish to constrain developers to writing parallel code in PVM.

Codine

Codine [81] by Genias software, is a widely-used commercial resource management software system. It has the ability to not only manage physical computational resources such as memory and disk space, but also software licenses. It supports check-pointing and migration of user jobs, and is accessible through a user-level API. Although the level of functionality is greatly increased over other batch queueing and wide-area management systems, the architecture of Codine is queue-based, as found in many other batch queueing systems such as DQS, NQS and PBS. It does, however, handle wide-area clusters. There must exist a queue master, which performs the scheduling of user requests onto computational resources, and each machine comprising the cluster must execute a daemon. Jobs are scheduled according to a first-in first-out (FIFO) scheme, and the cluster administrator can determine the way in which jobs are placed onto resources. The system allows dependencies to exist between jobs submitted to the queues. Parallel execution packages, such as PVM and MPI are directly supported. When the user submits a request in which the program uses one of the supported parallel packages, multiple resources are co-scheduled while

the job executes. There is no direct support for multi-threaded jobs to be split across resources.

Codine was later merged with other tools, to form the Global Resource Director (GRD) [82]. Directed at critical resource management issues, GRD was initially released for the Cray and SGI platforms. Codine provides job management capabilities, while PerfStat [83] provides performance monitoring capabilities. An advanced scheduler, GDS, is used to implement multiple scheduling policies such as functional priority (by user, department, job class or project), share-based (user share tree or project share tree) and urgency-based (initiate time or deadline) policies [66]. In addition GDS allows the system to be temporarily overridden when the needs arise. GDS allows dynamic scheduling of user jobs and utilisation management or resources.

Condor

Condor [153] is a distributed batch system developed at the University of Wisconsin-Madison from 1988 to execute long-running jobs on workstations that are otherwise idle. The emphasis of Condor is on high-throughput computing. The Condor system recognises that not all users utilise their systems all of the time, and that while they are not using their systems, they may be used for other processing tasks. While Condor provides the user with a uniform view of processor resources, thus making remote access to a compute resource easy, it also guarantees the workstation owner will receive the immediate response of their workstation, should they wish to use it.

Condor provides the user with a uniform view of processor resources. It also guarantees that the user's program will be run on an unloaded machine. Condor also allows users' jobs to be run on any machines in the processor pool, whether or not the user has a valid account on the target machine.

Scheduling and resource management in Condor is achieved through the use of matchmaking [192]. Matchmaking is the way in which Condor addresses the fact that in metacomputing environments, it is highly likely that resources will be owned by different organisations or institutions, and each one of these has its own usage and management policies. For example, a management policy may state that a particular resource is only available to core staff between certain hours, but is to be available to all bone fide Condor users outside of those hours. Matches are made using a semi-structured *classified advertisements* data model, which is used to make queries and publish services. A designated matchmaker is used to match queries with services and inform the entities of the match.

Computer Center Software

Based at the University of Paderborn, the Computing Center Software (CCS) [194–196] project is concerned with the resource access and allocation problems in metacomputing environments. It offers the user the option of running jobs in either interactive or batch mode.

Scheduling within CCS is performed via an Implicit Voting Scheme (IVS) [193] coupled with a priority system to express urgency of tasks. If the system is not congested, a simple first-come first-serve (FCFS) algorithm is used. The FCFS algorithm is explained further in section 4.3. Once the system becomes congested, however, the job mix is inspected and if most of the jobs are batch programs, a first-fit decreasing-height (FFDH) algorithm, which is an instance of the bin packing algorithm, selects the first available job that takes the greatest number of processors. Tasks are assigned so as to maximise the average utilisation of the processors that comprise the metacomputer. If most of the jobs in the request list are interactive, another algorithm, first-fit increasing-height is used, which gives preference to shorter jobs and reduces the average waiting times. One of the significant contributions of CCS was the development of a resource definition language, (RDL) [22] which targets and specifies the types of interconnections between reconfigurable Transputer-based parallel computers.

2.2 Middleware and Metacomputing Systems

The primary purpose of a resource management system is simply to provide fair, efficient use of shared computational resources. Users are unwilling, and sometimes unable, to understand and implement the framework associated with fault tolerant distributed systems. This is further discussed in appendix A. One of the fundamental purposes of metacomputing environments is to allow larger and more problems to be solved without the need for detailed technical knowledge of the computers that provide the computational services. Although one of the mechanisms for providing this is through resource management systems, they do not provide enough functionality to be classified as metacomputing systems.

One of the fundamental differences between cluster computing and metacomputing is the nature of the job which is to be run on the system. Although some cluster computing environments support fine-grained parallelism through the use of a third-party system such as PVM [79], MPI [68], p4 [32] or Linda [80], most cluster

computing environments do not support single jobs that comprise of a number of largely independent tasks that can be run in parallel. In those systems that do, this is achieved through the use of a shell-script in which the programs to be run are contained. Upon execution, the separate programs are executed by the target machine as if they were a single program. Although batch queueing systems and wide-area resource management systems have provided good solutions to the problem of managing a cluster of workstations, they fail to scale up to thousands of machines, which may dynamically join and leave the extended cluster. They are also unable to manage and reason about the re-use of results. We characterise metacomputing environments as having the ability to process a request for execution that comprises of independent tasks, and schedule each as separate requests, organising task intercommunication appropriately. In addition, metacomputing environments have the ability to re-use results, and sometimes partial results, in further computations.

Baker and Fox point out that cluster management systems and distributed cluster control environment software either come as a software layer positioned between the native operating system and user applications, or as a partial or complete replacement for the native operating system [20]. Metacomputing environments, too, can either be positioned as a software layer above the operating system, or as a replacement for the operating system, from the application's point of view.

Middleware is a layer of software that provides high-level services to applications, abstracting over low-level details that may differ between platforms. This software layer allows multiple processes running on one or more machines to interact transparently across a network. The relationship between middleware, metacomputing and traditional computer architecture is shown in figure 5. Middleware exists on top of the host computer's operating system, and in some cases takes over most of its functionality [95]. Enabling technologies include DCE [197] and CORBA [177]. User queries are made to the middleware to execute applications on behalf of the user. The most typical example of this is a remote execution request. Middleware tools are described in the next sub-section. Examples of middleware tools are described and discussed.

Metacomputing systems make use of the services provided by middleware, which focus on larger-scale distribution and larger problems. There are a number of metacomputing systems in existence [11,26,39,61,69,89,95,113,158,200]. Some well-known metacomputing systems are Globus [69] and Legion [95]. A review is given in [19], which provides a description of a number of metacomputing systems, as well

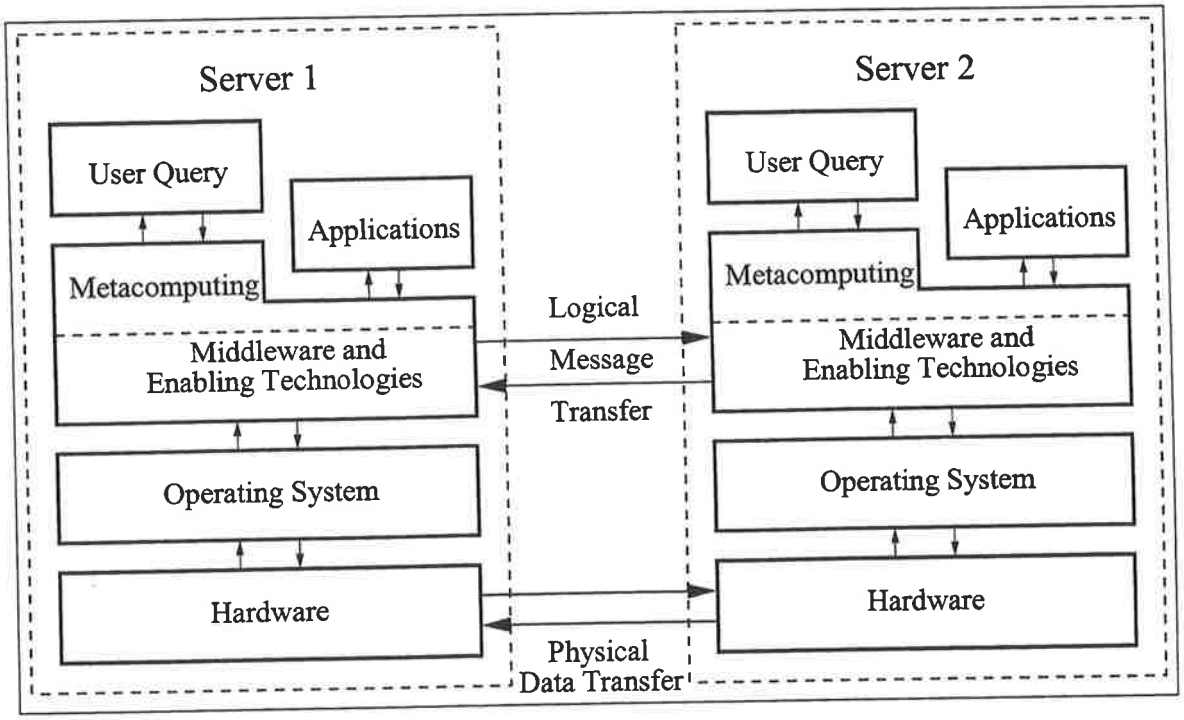


Figure 5: The relationship between middleware software and modern, large-scale computing. Metacomputing systems can be constructed to use the services provided by middleware. In this diagram, user queries are placed slightly higher than applications to indicate that the applications are used to satisfy user queries. Enabling technologies include those which provide naming and directory services, such as DCE [197] and CORBA [177].

as what may be termed middleware tools. Middleware tools are tools or systems that are independent of a particular metacomputing system, but are designed to run in the framework provided by a complete system. The Globus and Legion systems recognise several general characteristics of metacomputing systems, some of which may be mutually exclusive in a real implementation:

Scale and the need for selection. There is need for metacomputing environments to be scalable to large numbers of distributed machines. The system should also have the ability to sensibly choose to use a particular machine, or group of machines, in preference to others.

Heterogeneity at multiple levels. With the proliferation of different computer architectures and interconnection network architectures, systems must be able to adapt in order to make the best use of such heterogeneity. It follows that the

software that runs on the distributed machines must also be capable of being heterogeneous from the point of view of machine usage policies and the software that provides high and low-level services.

Unpredictable structure. Unlike traditional supercomputers, it is not possible to completely specify in advance what the processor topology or interconnection network characteristics will be when running a processing job. For this reason, metacomputing environments must be able to withstand changes in environmental characteristics.

Dynamic and unpredictable behaviour. In contrast to supercomputers that may run a single processing job at a time, it is almost a certainty that users will be sharing resources in a metacomputing environment, from processors, interconnection network bandwidth, to I/O devices.

Multiple administrative domains. The problem of allowing a user to run their code using a different site introduces issues of security and authentication to the general problem of metacomputing, as does to issues of charging for system time, and access policies for users of the metacomputing environment.

A selection of metacomputing projects, representative of the state-of-the-art, is presented in subsections 2.2.2 through 2.2.7. Our metacomputing project, DISCWorld, is introduced in section 2.3. DISCWorld is compared against other metacomputing projects in section 2.4, and the chapter is concluded in section 2.5.

2.2.1 Middleware Tools

In this sub-section we discuss tools and systems that, while not recognised as metacomputing systems in their own right, are designed to run in a metacomputing framework, or to provide some of the functionality of a metacomputing system.

There exist a range of tools that facilitate the writing of parallel and distributed programs, such as the MPI-CH [97] implementation of the Message Passing Interface standard [98] and PVM [79]. Both systems present the user with a message-passing environment which may be composed of a heterogeneous collection of computers. While not true metacomputing systems, these tools have played a large and important part in increasing the popularity and accessibility of parallel and distributed applications. MPI-CH and PVM provide an application programmer interface (API) and run-time communications library against which a programmer

can write and execute their applications. PVM additionally provides a run-time environment with which the user can interrogate the virtual machine, add or delete hosts, and control running parallel jobs. The machine is independent of any single parallel program execution. In contrast, the virtual machine created by MPI-CH exists for the duration of the currently-running parallel program only; additional nodes are unable to be requested or released at run-time. In both systems, fault tolerance and reliability issues must be addressed by the user. Programs such as PLUS [31], which are built over an existing message passing library such as PVM, re-produce some functionality of a metacomputing system; and some projects emulate parallel computers with off-the shelf systems, such as the Berkeley NOW project [12], which schedules parallel workloads across clusters of computers [16, 52].

DCE

Distributed Computing Environment (DCE) [180, 181, 197, 198] from the Open Software Foundation (OSF) is one of the most influential distributed computing products on the market. DCE provides services and tools that support the creation, use and maintenance of distributed applications in a heterogeneous computing environment. It is a standard set of tools, such as DCE RPC and DCE Threads, and services, such as the DCE Directory Service, DCE File Service, Security Service and Distributed Time Service. Each of the tools and services are integrated with the aim to provide interoperability and portability across a heterogeneous collection of platforms.

AppLeS

The Application Level Scheduler (AppLeS) [24] project is an application-level scheduling system developed at the University of California, San Diego from 1996. AppLeS is not a resource management system; it is what is commonly called middleware. It interacts with metacomputing systems as Globus [69] and Legion [95] or cluster management applications such as PVM [79] and MPI [68]. AppLeS is an agent-based methodology for application-level scheduling. AppLeS agents are based on the application level scheduling paradigm, where everything about the system is evaluated in terms of its impact on the application [25]. Each application has its own AppLeS, and each AppLeS combines both static and dynamic information to determine a customised application-specific schedule and implement that schedule on the distributed resources of the metacomputer.

AppLeS relies on application-specific and system-specific information to produce good schedules. In this context, *good schedules* are judged by the application's performance criteria. Schedules derived from predictions of application and system state are only as accurate as the predictions themselves, and because the whole system is dynamic, the predictions have a finite lifetime, beyond which they become out of date.

The AppLeS agent has a single active agent, called the coordinator, and four subsystems: the resource selector; the planner; the performance estimator; and the actuator. The resource selector chooses and filters different resource combinations for the application's execution. The planner generates resource-dependent schedules for given resource combinations. The performance estimator generates performance estimations for candidate schedules according to the user's (or application's) performance metric, and the actuator implements the best schedule on the target resource management system [25].

Information, contained within the AppLeS information pool, is supplied to the AppLeS agents in a number of ways. The Network Weather Service [237] provides CPU load predictions and network statistics for the period in which the application will be scheduled. Through the user interface, the user provides specific information such as the structure and characteristics of the application, performance criteria, execution constraints, and any login information that may be needed to use remote machines. Performance estimation and resource selection information is gathered by any default models that have been built up through use of the metacomputing environment.

Available information is used to filter infeasible resource sets from the resource pool. Remaining resources are then prioritised according to an application-specific notion of distance between resources, and promising sets of resources are identified by the resource selector. The planner is then invoked to propose a schedule, which is evaluated by the performance estimator. Once the best schedule, according to the application's performance criteria, has been identified, the actuator implements the schedule.

MARS

The Metacomputer Adaptive Runtime System (MARS) [78] from the University of Paderborn is a metacomputing framework that aims to solve a number of problems concerned with the problem of running parallel codes on a metacomputer: how

to determine a good initial task-to-processor mapping; how to cope with changing network performance; how to cope with varying node performance (e.g. concurrent usage); how to migrate tasks on heterogeneous systems; and how to cope with processor/network failures or extensions.

The runtime system uses application-specific data, such as communication characteristics, processor utilisation and program phases, and system-specific information such as node performance, network throughput and latency to determine where to run, and whether to migrate tasks between heterogeneous processors. MARS uses application- and system-specific information to predict applications' future resource utilisations in an effort to improve task migration. Migration decisions are made by a *Migration Manager*. MARS is currently written in C and uses MPI; a preprocessor inserts MPI calls to facilitate the collection of statistics by the runtime system.

MARS has two fundamental instances: Monitors, which collect statistical data; and Managers, which use the statistical data in order to make initial task placement and task migration decisions. Due to the fact that not all machines are binary-compatible, task migration is restricted to certain places in a computation; these places are identified by the addition of certain MPI calls, which are manually added. User's code is linked to the MARS runtime library, which allows the Monitors to intercept send and receive calls and generate application and system statistics.

Task dependency graphs are generated by the Application Manager, which are consolidated by a Program Information Manager. Nodes in the dependency graphs represent serial portions of code between a pair of send and receive calls, and are called Independent Blocks (IBs); edges denote the precedence relationship between IBs. Consolidation is necessary as task may generate different dependency graphs on subsequent runs. Dependency graphs of multiple task runs are consolidated to produce a graph of the likely communication patterns of a given task, which will aid in the placement of the tasks onto processors.

It is remarked that although most applications benefit from the use of the dependency graph in task placement, it may be difficult to construct for applications with highly irregular communications structures. The statistical variance of the dependency graph is used to modify the decision algorithm. The authors also justify the use of another system call when issuing a communications send or receive by the knowledge that the network latencies are an order of magnitude higher in WAN environments than tightly coupled parallel systems [78].

Nimrod

The Nimrod [1, 2] project addresses the problem of performing a large number of parameterised simulations on a set of distributed computers, each simulation having a different parameter set. It does not address the problem of parallelising an individual program, or allow a series of programs with interdependencies to be executed in parallel, nor does it address the issue of fault tolerance. The original implementation of Nimrod required all participating resources to run DCE [198].

Resource management in Nimrod is handled by whatever queueing system is present on the computer that is chosen to run the jobs. Nimrod submits the jobs that it generates to the target computer's queueing system, where it is executed in turn, and the results returned to Nimrod. The decision to let Nimrod submit jobs to the target's queue management system, rather than actively support the distribution of jobs was an effort to reduce the complexity of the application program. It is pointed out in [146], that the skill-set of programming in truly distributed environments is different to that found in ordinary application programming.

The main contributions that Nimrod has made to the field of metacomputing are those of job transfer and the user-centric view of the metacomputing system. Nimrod takes care of the job transfer on behalf of the user by sending the appropriate input files to the target processor via a remote file transfer server. It was also one of the first systems to provide a user-centric view to the computational environment.

Recently, a new version of Nimrod, Nimrod-G, is being ported to operate under the Globus metacomputing environment [3]. It is intended that the new version will utilise the functionality of Globus that allows the user to specify time and cost constraints on experiments.

2.2.2 Globus

Globus [69, 70] is a metacomputing environment under development at Argonne National Laboratories and the Californian Institute of Technology. It consists of a metacomputing infrastructure toolkit, which provides basic capabilities and interfaces for communications, resource location, scheduling and data access [69]. Each toolkit component has a well-defined interface which, in combination, define a metacomputing abstract machine, upon which higher level services can be built.

The Globus project does not aim to re-invent existing technologies such as PVM [79], MPI [98], Condor [153] or Legion [95], but provides basic infrastructure

through the development of low-level mechanisms that can be used by higher-level services. The project also aims to provide techniques that allow such services to observe and guide the mechanisms. Some of the lower-level mechanisms are: resource location and allocation, communications, unified resource information service, authentication interface, process creation, and data access.

The unified resource information service contains information about the status of the system, for example: static characteristics of a processing node, instantaneous performance information and application specific information. This information is gathered by different sources, and can be accessed by a single mechanism within Globus.

Globus modules can be influenced in the decisions that they make by higher-level services. This is accomplished by the use of rule-based selection, resource property inquiry and notification mechanisms. Rule-based selection is used to identify strategies with which the low-level module can perform a given task. The resource property inquiry module requests information from the Globus unified information service, which contains the current state of the environment. The notification module allows a call-back mechanism between the higher-level service and the low-level mechanism such that in the event of an exception, the mechanism can notify the service of the event.

Resource management in Globus is achieved through the interaction of the Globus toolkit with any schedulers that may run on the local system. An extensible resource specification language (RSL) [48] is used to communicate requests for resources between components. The RSL is a simple language that allows the system to request resources containing characteristics embedded in the language. The RSL describes, principally, the physical machines on which to run programs. Local and global services that the programs use are considered to be fine-grained enough that they can be run on any of the machines in the computing environment.

The Globus resource management system revolves around the concept of resource brokers, software that acts as an interface and translator between higher-level specifications of requests, and more concrete representations of the request (in RSL). These brokers are application specific, having the ability to understand high-level requests from user clients. They progressively refine the client's request until it becomes expressible as a request for specific resources. After translating the specific requests into RSL, they are dispatched to the Globus Resource Allocation Manager (GRAM). The GRAM [48] provides the local component for resource management.

Each GRAM is responsible for a set of resources operating under the same site-specific allocation policy, often implemented by a local resource management system such as LSF [188] or Condor [153]. GRAM provides a standard network-enabled interface to local resource management systems. While individual sites are not constrained in their choice of resource management tools, the computational grid tools [71] and applications can express resource allocation and process management requests in terms of a standard API.

Resources and computation management services are implemented in a hierarchical fashion. An individual GRAM supports the creation and management of a set of processes on a set of local resources. A computation created by a global service may then consist of one or more jobs, each created by a request to a GRAM and managed via management functions implemented by that GRAM. To implement a global directory, Globus uses a Metacomputing Directory Service (MDS) [67], which is based on the Lightweight Directory Access Protocol (LDAP) [230].

2.2.3 Harness

Harness [158] is a metacomputing framework designed for dynamic reconfigurability. A collaborative project between Emory University, UTK and ORNL, it is based on IceT [89] from Emory University.

Dynamic reconfigurability within Harness is achieved through the use of a "plug-in" mechanism by which services and computational resources can be added to and removed from the system [159]. The model describing the Harness distributed virtual machine consists of four layers: the abstract distributed virtual machine (DVM); heterogeneous computational resources; services; and, applications and users. Applications and users are able to utilise a consistent baseline, to which the DVM conforms.

Services consist of an interface specification; instances of the services, *plug-ins* are plugged into the DVM to provide processing capabilities. Services have attributes such as their shareability, exportability and their threadability. Shareability refers to a service's ability to be used simultaneously by multiple applications; exportability determines whether different DVMs are able to access the service; and threadability refers to the ability to execute multiple instances of the service simultaneously in a single DVM. Services are divided into three classes: kernel level services; basic services; and specialised services. Kernel level services are crucial to the operation of the DVM; they must be portable across the DVM. Basic services are those very

common services, which may or not be exportable. Specialised services include those that have non-standard platform requirements (eg very large memory requirements or massively-parallel vector processors) or performance requirements which restrict their execution on every machine in the DVM. As Harness is based on IceT, the remainder of this section describes IceT.

The IceT model of resources consists of multiple clusters of virtual environments belonging to multiple users, each with distinct levels of security and accessibility. Written in Java, native IceT processes and data are transportable. They can be uploaded to remote locations, and in the case of processes, can be executed without regard for the remote architecture or file system structure.

Fundamental to the IceT process model is the concept of *process spoking*, which refers to the ability of a process and its data to be uploaded to a remote computational resource for execution, as opposed to the common model of remote requests and responses. The main difference is that when a process is uploaded to a remote resource, it can exert independence from the requesting resource (unlike the CORBA and RMI models), and can manage the uploading of any other processes that are necessary for the computation or maintain persistent communications with other processes.

IceT is implemented in Java. Java servers allow byte-codes to be executed on remote nodes where the requesting user does not have normal access privileges. This is achieved through Java's portability and security measures, and the uniform, system-independent view of the underlying architecture. Java's *just-in-time* features are extensively relied upon to achieve better execution performance than pre-compiled Java byte-code running in a standard Java Virtual Machine.

Virtual environments are created by the user adding hosts into the configuration held by the local daemon. Remote virtual environments can be added, which causes any hosts contained within the remote virtual environment to become accessible through the local daemon.

The architecture and operation of Harness and IceT are similar to PVM. In order to execute pre-compiled IceT Java byte-code, as in PVM, the user must either run the program from the command-line or spawn an IceT task from a local daemon's console. Providing the node's security manager allows the byte-code to be executed, it is sent to the target node, and is parsed for dependencies on other byte-codes (Java packages) or shared system-dependent libraries. In the case of dependence on other byte-codes, they are uploaded to the target host, and in the case of shared libraries, if

there is a version of the library that is available for the architecture, it is uploaded [90]. No indication is given in the literature of the behaviour when a shared library of the appropriate architecture is not available. No mention is made of the method by which scheduling in either Harness or IceT is achieved.

2.2.4 Infospheres

Infospheres [36–39] is a project from Caltech that addresses dynamic scalable distributed systems. The research is concerned with developing infrastructure to support structuring distributed applications by composing components using sequential, choice and parallel composition. It is implemented in Java [88] and TCP/IP, for use over the World-Wide Web (WWW). The project is one of the first to approach the metacomputing problem from the point of view of composing existing, well-characterised components together to perform a task. Infospheres is not intended to support seamless parallelism, high-performance computation or fault tolerant transactions.

Compositional units are abstractions of processes and sessions. Processes can be composed in parallel. Sessions are collections of processes composed in parallel. Sessions may be composed using sequential and choice composition. The infrastructure supports distributed applications, which can be structured by nesting processes and sessions. The object model is state-based, where every object has a persistent state for the lifetime of its corresponding entity. The execution of the system is regarded as a set of states where state is assignment of values to a given set of variables.

Interprocess communications is achieved by the use of asynchronous RPC [27]. Incoming and outgoing message queues are local to each process, created and destroyed at will. Queues may have different priorities. Output queues may be bound to an arbitrary number of input queues and messages are sent in a FIFO order. When multiple output queues are sent to an input queue, the result is a fair merge of the sequence of messages. Input queues are typed in the messages they are allowed to accept.

Parallel computation is implemented by an appropriate binding between input and output queues. Processes may be mobile over the network, but only between sessions; they must be immobile within sessions. Sessions are instances of applications, implemented as networks of processes. Processes and sessions have *specifications*, which are precise definitions of the behaviours. Infospheres

supports three specifications: process specifications, interface specifications and session specifications.

Finding specific processes is done using standard web technology: processes are found by looking up the appropriate home page. Finding an appropriate process type is harder: the type specification must be clear enough to identify and compose processes; and an arbitration scheme must exist if the interfaces of two types do not match. Distributed objects are compared by checking to see if their interfaces match.

The target applications of the Infospheres project are collaborative applications, such as distributed design frameworks and distributed calendars. One of the features of the Infospheres system is the provision for long-lived collaborations by the incorporation of the idea of persistent components that may, when not actively undergoing computation, have their state serialised and be stored on a persistent media, only to be de-serialised when needed again.

Resources in the Infospheres environment are strictly computational [191], and clients are expected to reserve resources in order to be able to use them. To facilitate this, clients' systems can send Java *agents* to remote machines in order to request a resource be reserved. Agents are directed to resource managers, which act as brokers for the resources that they control.

2.2.5 Legion

Legion [95] is a research project aimed at providing a highly usable, efficient and scalable system, based on solid object-oriented principles. The output of the project is a single, coherent virtual machine that addresses the issues of: scalability, programming ease, fault tolerance, security, site autonomy and has an extensible core [94]. Legion is similar to a number of other projects, namely Nexus [73], Castle [46], NOW [12] and Globe [118].

Legion achieves multiple language interfaces and interoperability through object wrappers for legacy codes, similar to those used by CORBA [177] and DCE [197]. As well as providing wrappers for legacy objects, the Legion designers achieved widespread use through allowing the system to be the target of compilers. High performance is achieved via two methods: resource selection, and parallel computation. Resource selection is performed using resource availability and affinity, which is an extension of Condor [153], DQS [91] and LoadLeveler [123].

Written in a parallel variant of C++ called Mentat [92,93], Legion is an attempt to create a single nationwide metacomputer using loosely coupled workstations. Legion

was designed to transparently schedule application components on processors, manage data transfer and coercion, and provide communication and synchronisation in such a manner as to minimise execution time via parallel execution of the application components. Legion is based on Mentat [92], an object-oriented parallel processing system.

It is intended that Legion be used almost completely for parallel applications. The concept of *resource transparency* is used, where the user (and application program) do not depend on a certain number or type of processor. Latency-tolerant, relatively large-grained parallelism is targeted. Object wrappers are provided for parallel components, and Legion supports parallel method invocation. The Legion runtime system is exposed as an 'open system', and has an associated message-passing API.

Memory within Legion is treated as a single, persistent object space [96]. Legion does not make resource allocation decisions, but provides the basic mechanisms needed to make informed mapping decisions between resource objects and carry out these mapping decisions. If an object needs to contact another object, and the target object does not already exist, it is created, as is described in [137].

Fault tolerance is addressed with the knowledge that in a truly distributed heterogeneous system, hardware and software failures will be routine, and that each physical resource and application will have its own concept of what is necessary. Knowing that writing programs to achieve fault tolerance is a difficult and error-prone task, Legion does not mandate policies, but instead applications can select the level of fault tolerance they require. Legion facilitates this by encapsulating fault tolerance protocols in base classes, which may be extended by users. Legion implements different levels of fault tolerance, depending on the penalty that the application-writer is willing to pay.

Scheduling data parallel components in Legion is static, and is broken into three distinct phases: *processor selection*, *load selection* and *placement*. First, candidate processors are identified. Secondly, the number and type of processors are used and the data domain is decomposed. Lastly, tasks are mapped to chosen processors so that communication time is reduced.

Objects create *mappers* and provide them with a description of the particular placement problem. The mapper then marshals the objects that will be involved and asks the system for a snapshot of the current system state (i.e. available resources). It then makes the appropriate decisions, based on a heuristic search of the solution space and passes the decision on to a module that tries to implement the decision,

the *implementor*.

Mappers perform the function of taking a high-level specification of a placement problem and converting it into a list of possible placement decisions. Although Legion puts the onus of supplying a default mapper on to the application writer, if other mappers are available, the user can choose in order to personalise the selection process.

Legion also has the concept of a *jurisdiction magistrate*, which contains and enforces the local policies. It is this object which handles all placement failures, security breeches and querying of placement decisions.

Programs are represented in Mentat by graphs (or DAGs), and parallel execution is based on a macro data flow model, where edges in the graph denote dependence relations between nodes, representing operations (or actors). In Mentat, *future lists* are sent out with actors, so that actors are aware of the nodes on which the output data depends. The data dependency is very fine-grained, operating at the instruction-level.

Parallelism is encapsulated between objects by allowing subsequent actors to use the results of previous actors, which may have not yet been computed. Coherency of variables that are to be shared is maintained by enforcing single-assignment of future variables. Thus, but the use of futures, parallelism opportunities are exploited without the danger of variables being modified between uses. The run-time system detects data dependencies and organises task scheduling.

2.2.6 DOCT

Distributed Object Computation Testbed (DOCT) [200] is a collaborative project between the San Diego Supercomputer Center (SDSC), Caltech, NCSA, Old Dominion University, Open Text Corporation, Science Applications International Corporation, University of California at San Diego and the University of Virginia. DOCT is a large metacomputing framework being constructed using Legion and AppLeS.

Designed to manage a tera-byte sized persistent document handling system [162], the software included in DOCT includes HPSS [122], MDAS (from SDSC), Legion, AppLeS' Network Weather System and IBM's DB2 parallel object-relational database. The system features intelligent agents which will be able to perform many diverse actions. Intelligent agents use the Legion framework as a basis for distributed objects and Nexus is being considered for distributed communications. Resource usage is coordinated by the AppLeS scheduling system.

2.2.7 WebFlow

WebFlow from Syracuse University is a metacomputing project which aims to produce a general-purpose Web-based visual interactive programming environment for coarse-grain distributed computing [26]. The project aims to provide coarser-grained units for Java distributed computing than the Java class. WebFlow is a programming paradigm implemented over WebVM. WebFlow is part of a larger project, WebSpace [174] to create a Web-based collaborative environment.

WebFlow is implemented over a mesh of WebVM servers. WebVM servers are implemented using Jeeves [219], which is a Java Web server. WebFlow allows modules (or servlets, atomic encapsulations of WebVM computation) to be run on demand, supports communication between modules, and allows users to create and destroy applications (sets of interconnected modules).

WebFlow is very similar to Infospheres in its support for code modules, communication between modules, and sessions. In both systems modules communicate by the use of ports. Sessions define the application that a user creates using modules. There is very little information available on the development and internal workings of the WebFlow system.

2.3 DISCWorld

Distributed Information Systems Control World (DISCWorld) [113] is a prototype metacomputing model and system being developed at the University of Adelaide. The project was started in 1996. The basic unit of execution in the DISCWorld is a *service*. Services are pre-written pieces of Java software that adhere to an API or legacy code that has been provided with a Java wrapper. Users can compose a number of services together to form a complex processing request.

The DISCWorld architecture consists of a number of peer-based computer hosts that participate in DISCWorld by running either a DISCWorld server daemon program or a DISCWorld-compliant client program. DISCWorld client programs can be constructed using Java wrappers to existing legacy programs, or can take the form of a special DISCWorld client environment which runs as a Java applet inside a WWW browser. This client environment can itself contain Java applet programs (Java Beans) [58] which can act as client programs communicating with the network of servers. The peer-based nature of DISCWorld clients and servers means that these servers can be clients of one another for carrying out particular jobs, and can

broker or trade services amongst one another. Jobs can be scheduled [128] across the participating nodes. A DISCWorld server instance is potentially capable of providing any of the portable services any other DISCWorld server provides, but will typically be customised by the local administrator to specialise in a chosen subset, suited to local resources and needs. The nature of the DISCWorld architecture means that we use the terms client and server somewhat loosely, since these terms best refer to a temporary relationship between two running *programs* rather than a fixed relationship between two host *platforms*.

DISCWorld is targeted at wide-area systems and applications where it is worthwhile or necessary to run them over wide areas. These will typically be applications that require access to large specialist datasets stored by custodians at different geographically separated sites. An example application might be a land planning one [43], where a client application requires access to land titles information at one site, and digital terrain map data at another, and aerial photography or satellite imagery stored at another site. A human decision-maker may be seated at a low performance compute platform running a WWW browser environment, but can pose queries and data processing transactions of a network of DISCWorld connected servers to extract decisions from these potentially very large datasets without having to download them to their own site. This is shown in figure 6.

Our vision for DISCWorld is an integrated query-based environment where users may connect to a "cloud" of high performance computing (HPC) resources (where the heterogeneity and exact composition of the cloud is hidden from the user) and request data retrieval and processing operations. Users themselves may only be connected into the cloud by a low bandwidth network link such as that provided by a modem line. This action-at-a-distance query-based approach appears an appropriate one for decision support applications where the user is provided with a collection of application components that run *in situ* in a WWW browser and help him control remote HPC resources. Much of our research to date has considered the multi-threaded software daemon (DWd) that runs on each participating DISCWorld service providing host.

DISCWorld is aimed at providing high-performance services, with which non-specialist scientists can compose challenging, big, and complex problems for solving over a distributed collection of possibly heterogeneous computers. One of the design goals of the DISCWorld metacomputing infrastructure is that any parallelism that can be extracted from the problem that the user has posed will be seamless to the user.

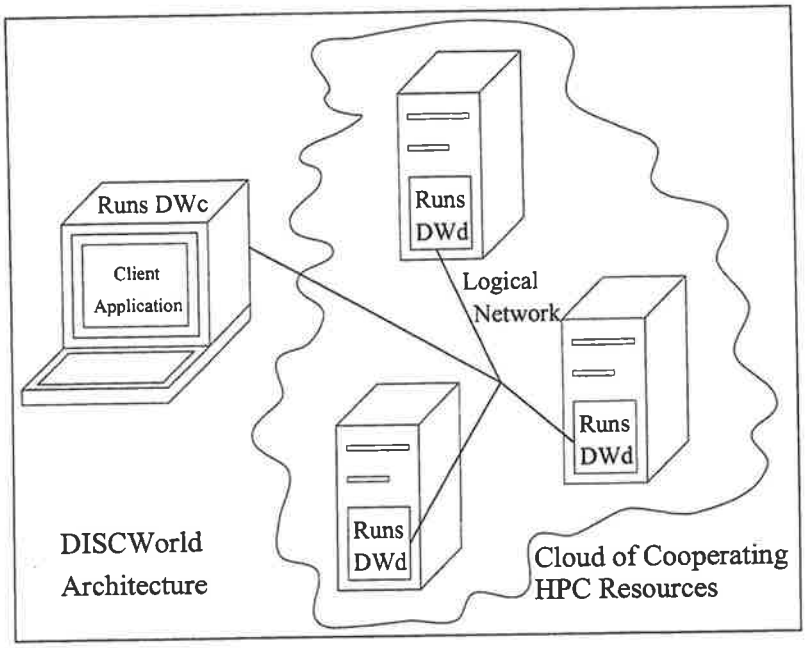


Figure 6: Conceptual view of DISCWorld architecture. A number of cooperating server hosts communicate and share work, brokered by the DISCWorld daemon. Each is capable as acting as a gateway for client programs running on server hosts or as specialised graphical interface clients.

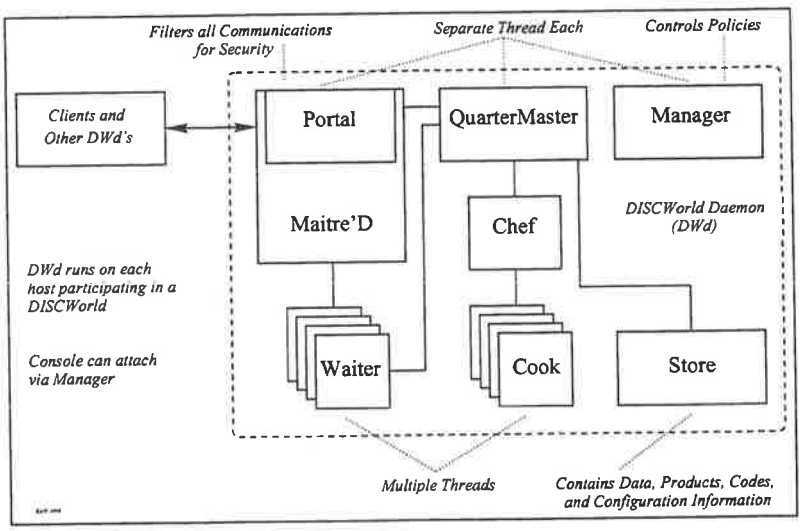


Figure 7: Detailed component breakdown of DISCWorld daemon. Code modules are represented by boxes; modules executing in separate threads are shown.

This approach differs somewhat from Infospheres, the only other research project that has service-, or *component*-based approach, in that Infospheres is designed to facilitate collaboration between users, and allow programmers to design and build distributed system components.

When user queries are submitted to the DISCWorld system, they are decomposed into services; services are the items which are scheduled in the DISCWorld. Data and services may be moved to the host at which the least cost is found. This thesis uses DISCWorld as an implementation framework.

2.4 Discussion and Comparison of Metacomputing Systems

DISCWorld has a constrained objective, with less ambitious goals than the Globus and Legion systems. The notable difference between these systems and DISCWorld is that most users of our system will not be programmers, *per se*, but will be merely interested in the output of a service request.

DISCWorld is a high-level, object-oriented system that uses Java as an implementation vehicle. As DISCWorld does not directly address fine-grained parallelism, resources do not need to be co-scheduled as in Globus, but services are expected to run in a reasonable amount of time. In addition to scheduling frameworks, some metacomputing approaches favour reserving the resource to be used [72,191].

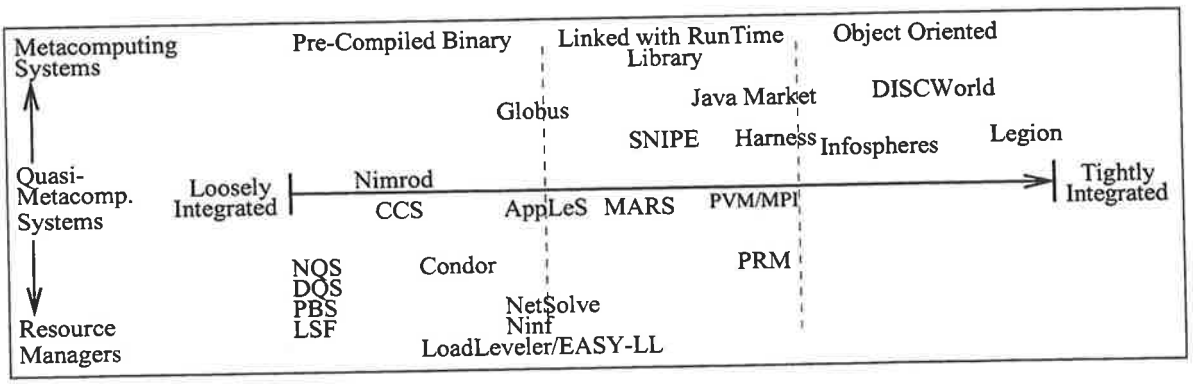


Figure 8: Metacomputing systems, middleware tools and resource manager systems integration with user processes.

The DISCWorld project addresses the issue of *constrained* metacomputing, by making the decision that only restricted datasets and pre-defined operations on data will be allowed. As such, all functions that can be applied to data will be written by developers; end users will not have to concern themselves with the implementation details. This means that the services that a user can request are constrained to those which have been defined and written for the DISCWorld system. This is in contrast to unconstrained systems, where users can submit arbitrary binaries for execution, such as Globus.

We believe that being *constrained* is an important characteristic of a metacomputing system for a number of fundamental reasons. Allowing users to submit arbitrary binaries is a security hazard for fear of malicious attacks, and the lack of control over what information is being collected and disseminated by user programs. We are not able to mandate that all users must submit source- or byte-code that can be analysed for performance characteristics or malicious code. Therefore, if users can submit their own binaries, the task of characterising programs which may only be run once or twice, in order to make intelligent scheduling decisions, is very difficult. Together with the benefit of being able to characterise services more easily and effectively, pre-written services, with knowledge of the DISCWorld, can be written to take advantage of DISCWorld features and any available parallelism inherent in the high-level processing request, can be exploited.

Services share similarities with Infospheres *components*, especially in the way in which services are characterised. As mentioned in [37], the services are created using the methodology of component technology; services themselves are opaque, presenting the user with an interface specification, but giving no clues as to the service's implementation; they can have dynamic interfaces and undergo dynamic composition; and services can be selected from a world-wide pool. Services must either be written in a platform-independent way which supports reflection, or at the very least, a wrapper interface to the code must be supplied. Thus, it is up to the service writer to ensure that the services they make available to the system can be queried in a standard, platform independent way, even if the code that is actually executed is architecture-specific.

Most of the metacomputing systems discussed in this chapter, with the notable exception of Globus, are tightly integrated with the code that runs as a result of a user query. Tables 1 and 2 show the characteristics of the systems described in the previous section. This is represented pictorially in figure 8. Systems that define a resource as

either an object or an applet have user processes that are tightly integrated with the system. This typically allows more information to be collected by the system in order to make sensible object-placement or scheduling decisions. In none of the systems is the user able to directly control the placement of their processes; in virtually all cases, the system can make a better decision than the user.

Harness is similar in design philosophy to DISCWorld. Both projects present a constrained metacomputing environment. The main difference is that DISCWorld services are written by developers, and users simply use the services, while in Harness, the user is responsible for supplying their own programs. Harness and DISCWorld both recognise the need to be able to move service code and data. Harness does not perform any organised scheduling of service execution; the environment uses the nominated hosts in a manner very similar to that found in PVM.

Objects within the Legion system have the concept of *sovereignty*, which can restrict copying and movement. This is similar to the DISCWorld concept in which services and code may be restricted in movement.

2.5 Conclusion

A cluster has been defined as a group of machines that may be viewed as a single entity for the purposes of control and job assignment. In this chapter the topic of cluster control has been addressed from two viewpoints: that of resource management, and of metacomputing. There are fundamental differences between the objectives of resource management software and metacomputing environments.

Resource management software is designed to provide all users with fair access to the machines that comprise the cluster, while at the same time ensure that the machines are being efficiently utilised. Metacomputing environments, on the other hand, are designed to allow access to remote resources, where the term resources is able to be defined as anything from remote machines, data, processing services, or specialised hardware such as visualisation equipment.

We have presented summaries of a number of metacomputing projects, and have compared them with our prototype metacomputing environment, DISCWorld. The conclusion of this chapter is that although solutions to the resource management and metacomputing problems do exist, none of the solutions is directly applicable to the case in which a user does not have a great deal of knowledge about the system they are using.

System	User process integration with system	Parallel Job Support	Available System Information	User controls placement	Method of handling heterogeneity	Definition of a resource
DISCWorld	Tight	Y[a]	lots[b]	N	Java	objects[c]
Globus	Loose	Y[d]	lots[e]	N[f]	RSL[g]	MDS machines
Harness	Med	Y[h]	little[i]	N	Java'd PVM	machines
Infospheres	Tight	Y[j]	?	N[k]	Java	objects[l]
Java Market	Tight	Y[m]	lots[n]	N	Java	various[o]
Legion	V. Tight	Y[p]	lots[q]	N[r]	generators[s]	objects

[a]services can be implemented in parallel, unknown to the rest of the system
[b]system information is low, but job information is high
[c]services, results, nodes, networks
[d]co-scheduled tasks and parallel codes
[e]using MDS and architecture dependent software
[f]each application has a GRAM or broker
[g]RSL has architecture dependent information
[h]fine-grained PVM-style tasks
[i]PVM's load information
[j]serial, parallel object composition
[k]agent-based requests number and types of resources
[l]strictly computational objects
[m]individual applets independently run in parallel
[n]instructions per second, cost
[o]machines, applets
[p]object-based parallelism
[q]OS-level system information
[r]placement done by system mappers
[s]equivalent to Makefiles

Table 1: Summary of metacomputing system characteristics

The system that currently provides the most comprehensive metacomputing environment is Globus. However, this system requires users to supply binary programs to achieve their goals, and is aimed toward the execution of parallel programs. We feel that in order to make metacomputing a more accessible technology to those users that wish to pose high-level queries, the details of the system, especially how the programs they use work, must be hidden from them.

In systems targeted towards fine-grained parallelism, scheduling or process placement is a matter of co-scheduling available resources. The DISCWorld model features high-level services in a producer-consumer relationship, where the results of a service are available to be used as inputs to another. We do not address fine-grained parallelism directly; services can be implemented as parallel code but the service

System	User process integration with system	Parallel Job Support	Available System Information	User controls placement	Method of handling heterogeneity	Definition of a resource
AppLeS[a]	Med	Y	lots[b]	N[c]	N/A	machine
MARS[d]	Med	MPI	lots[e]	N[f]	MPI	machine
Nimrod[g]	Loose	Y[h]	v. little	Y[i]	binary	machine

[a]This is really a *scheduler* not a Resource Manager
[b]CPU load, network statistics and program characteristics
[c]each application has own AppLeS
[d]This is really a *scheduler* not a Resource Manager
[e]application specific data gained through preprocessor and historical information
[f]historical and predictive information
[g]not strictly a metacomputing environment. The tool is too high level – it is designed to operate *over* a metacomputing environment
[h]independent tasks submitted to a queue manager
[i]user chooses machine and queue to which jobs are sent

Table 2: Summary of middleware tools characteristics

interface presented to the DISCWorld is identical to the same service implemented as serial code. The task of scheduling DISCWorld services and data access is the focus of this thesis.

In the next chapter, we review some of the research concerning the placement of processes (jobs) onto nodes in a distributed system. In chapter 4 we present a simulation of placing and scheduling independent jobs across distributed heterogeneous processors. Chapter 5 introduces a general model that is used for scheduling in the prototype DISCWorld metacomputing system, and chapter 6 discusses implementation issues.

Chapter 3

A Review of Scheduling

Scheduling is a problem that is grounded in many different levels of computer science and computer hardware engineering. Various scheduling and sequencing problems have been addressed since the 1950's by researchers in computer science, operations research and discrete mathematics [64].

The scheduling problem in its most general form is known to be NP-complete [147], as is the creation of optimal execution schedules under a number of conditions. This is due to the large number of inter-related factors that directly and indirectly contribute to the execution time of the individual tasks. Consequently, many heuristics have been developed to generate adequate (but sub-optimal) schedules [56].

The problem that we consider is how to distribute (or schedule) processes among processing elements to achieve performance goal(s), such as minimising execution time, minimising communication delays, and/or maximising resource utilizations [34]. In a system consisting of real-time transactions, each of which requires computational, communication and data resources to processed, scheduling is the problem of allocating resources to satisfy the requirements of those transactions [142].

We recognise four main levels of instruction scheduling: machine-code instruction scheduling; interpreted program code (converted at run-time to machine-code); the scheduling of threads within a program; and the scheduling of programs within a distributed system. These are listed in order of increasing granularity and abstraction from the physical code that is executed by processors.

The systems described in chapter 2 operate at the last two levels: the scheduling of threads within a program; and, the scheduling of programs within a distributed system. This thesis is concerned primarily with the scheduling of coarse-grained programs across distributed systems.

In this chapter we review some of the research in the area of scheduling, across all levels of granularity. We provide discussion of the applicability of the research to scheduling coarse-grained programs. The problem of scheduling on a local and global scale is considered. Different models of scheduling, including static, dynamic and hybrid approaches are discussed in section 3.1. The scheduling of independent programs is discussed in section 3.2 and of dependent programs in section 3.3. A discussion of the state information that is available to schedulers is presented in section 3.4, and an historical review of scheduling is presented in section 3.5. The applicability of the existing research to the problem we consider is discussed in section 3.6. The review is summarised in section 3.7.

At the highest level, a distinction is drawn between local and global scheduling. The local scheduling discipline determines how the processes resident on a single CPU are allocated and executed; a global scheduling policy uses information about the system to allocate processes to multiple processors with the view of optimising a system-wide performance objective. Under the classification of global scheduling, the problem domain is further broken into the cases where the tasks to be scheduled are independent or related [7].

When executing a complex job in a distributed system, scheduling occurs in many places and on many levels. Proceeding in a top-down manner, the program is submitted into a queue of waiting jobs. At some point, the scheduler will select the job to run. The program (or current job) is broken into a number of tasks to be executed, each of which is scheduled, possibly on different machines. When each task arrives at the target machine, if it involves parallel computation, then each of those parallel components is then distributed to target processors. Once the code arrives onto the processor where it will be executed, it is then scheduled by the operating system. The scheduling of code onto a processor is done in conjunction with system processes and quite probably with other users' processes. Shown in figure 9, this is the manner in which most of the batch queueing systems discussed in chapter 2 operate.

Choosing the processors on which the tasks that comprise a program will run is a difficult decision. There are many factors, from the viewpoints of both the application and the system as a whole, that effect the decision, including: the number of tasks that comprise the program; the priority of the program; the current load of the system's nodes; whether all of the nodes can execute each task; and the resource owner's usage policies.

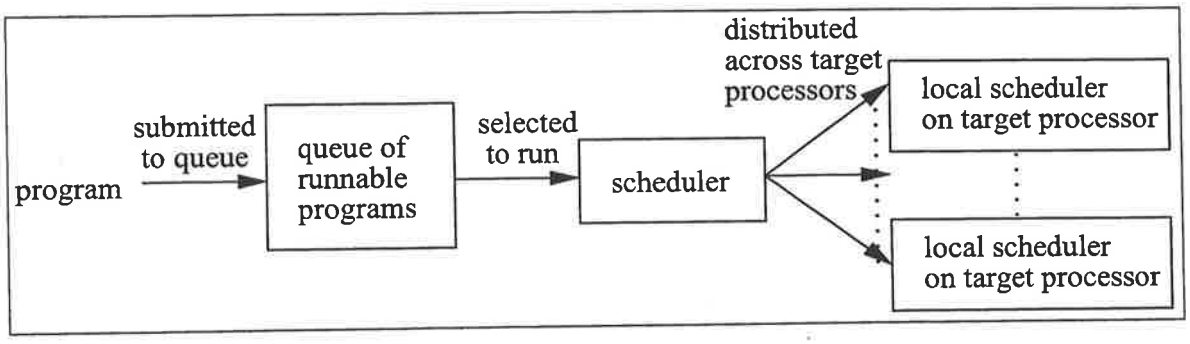


Figure 9: Graphical representation of scheduling. The user's program is submitted to a queue. When a program is selected to run, the scheduler allocates it to a target machine, where the local scheduler controls the local execution. If the target machine is has a parallel architecture, the program may be distributed across the local processors.

There are a number of reasons why scheduling programs, or the tasks that comprise the programs, is important to both the users and owners of machines. From the user's point of view, it is important that the programs they wish to run are executed as quickly as possible, using the resources that are best suited to the problem, which are also reasonably accessible to the user. In contrast, the machine owners wish to make the best utilizations of their resource. The term resource does not only refer to the machines that make compute cycles available, but also includes clusters of computers, communications link utilizations and storage services that may be offered, as well as whatever other peripherals each may have attached. The two objectives, fast turnaround time and optimal resource utilizations, are not always complementary. Owners are not usually willing to let a single user utilise all their resources, and users are not usually willing to wait an arbitrarily long time before they are allowed access to particular resources. Scheduling, from both points of view, is the process by which each party achieves a satisfactory quality of service, where the time spent waiting for a number of dedicated nodes on a supercomputer that is connected to a massive storage engine may be traded off against the actual opportunity to use the resources.

There is a complex trade-off between the user using resources that an owner may charge little (or nothing) to use, but may be of lower performance or have fewer desirable attributes (such as storage systems or high-bandwidth interconnections to other resources), and a larger resource which costs more to use but on which the problem will be solved more quickly and will be able to utilise any specialised

peripherals that are attached to the system.

There is no single set of terminology that is used throughout the scheduling literature. For example, the terms *program*, *job* and *task*, and the terms *host*, *processor* and *node* are often used interchangeably in the literature.

When we use the term program, we refer to code that is directly executable by the user of a system, such as a command shell or a compiled application. We use the term job to refer to a collection of programs that are run concurrently or sequentially, for a single purpose. An example of a job is a shell script that calls several programs. The input- and output-dependencies between the programs that comprise a job are sometimes termed the job's processing pipeline. We use the term task to refer to the code that runs on a single processor. The task may consist of a number of threads of control.

As multi-processor workstations become commodity items, a new distinction is being made between the term processor and the terms host and node. When only uniprocessor machines were available, the terms were equivalent; now one must be careful to use the correct terminology. We use the term node, or processing entity (PE), to refer to a processor which is not normally accessed by the user. For example, as the processors that comprise the parallel processing array in a Thinking Machines CM-5 are not directly accessible by the user, they are termed nodes. We use the term hosts to describe the processors to which users usually have direct access.

3.1 Scheduling Models

There are different approaches to the selection of processors onto which programs will be placed for execution. The models range from static, where each of the programs is assigned once to a processor before execution of the program commences, to dynamic, where a program may be reassigned to different processors before being executed. Finally, we describe a hybrid approach, which combines those taken by the static and dynamic models.

One of the first taxonomies of scheduling in distributed computing was performed by Casavant and Kuhl [34]. They divided the problem domain into static and dynamic scheduling. Static scheduling involves assigning the programs that comprise the job to processors and then not allowing them to move. This minimises turnaround time for the user but is not responsive to changes in the execution environment. Dynamic scheduling, or load balancing, involves the averaging of load over the

chosen processors, in an effort to maximise average resource utilizations and reduce turnaround time for the user.

Many theoretical studies consider off-line systems, and search for optimal solutions using the assumption that everything is known *a priori*. They often also assume nothing changes in the system's environment. Real systems, however, operate in a dynamic on-line environment, and need to contend with unpredictable arrivals of new work [62].

3.1.1 Static Scheduling

In the static model, every task comprising the job is assigned once to a (possibly different) node. Thus, the placement of a program is static, and a firm estimate of the cost of the computation can be made. Heuristic models for static task scheduling are discussed in [206].

One of the major benefits of the static model is that it is easier to program from a scheduling and placement view. The placement of tasks is fixed *a priori*; it is easy to monitor the progress of the computation and hence termination of processing is simplified. By the same reasoning, estimating the cost of jobs is simplified. Processors can give estimates of the time that will be spent processing programs. On completion of the program, the processor can be instructed to supply the precise time that was spent in processing. This allows the cost to the user to be updated, as well as any internal representations that are used for making performance estimates of new programs.

Using this model, it is also possible to reserve resources. Thus, instead of the scheduler that is creating the placement information simply looking up a (dynamic) table of host names and the programs available on those hosts, the system could send a reservation message to the host, enquiring as to its willingness to run a program.

When scheduling jobs in a static fashion, only the scheduler which creates the job placement and cost estimates needs to know the existence and relative costs of the other hosts in the system. Although other hosts may know about the programs that another host can execute, all they must know is how to send messages to the next (or previous) host in the job's processing pipeline. Using the static model, each host is told from where to get the input parameters that are needed, or where to send the outputs of the programs.

The static model allows for a 'global view' of programs and costs. Heuristics are used to decide whether to incur slightly higher processing costs in order to keep all

the programs involved in a job on the same or tightly-coupled nodes, or to search for lower computational costs and be penalised with slightly higher communication costs.

Unfortunately, the static model does not allow for the very real possibility that one of the nodes selected to perform a service may have failed, be isolated from the system due to network failure, or at least so heavily laden with jobs that it is not responding.

3.1.2 Dynamic Scheduling

General-purpose dynamic scheduling operates on two levels: the local scheduling discipline, and a load distribution strategy. The load distributing strategy determines how programs will be placed on remote machines. It uses an information policy to determine which information is to be collected from each machine in the processor pool, at what frequency, and also how often the local information should be exchanged with other machines. Scheduling may be approached from many different aspects, including the viewpoints of the user and the resource owner.

In a strictly dynamic scheduling model, the tasks that comprise a parallel or distributed job are assigned to processors based on whether a processor predicts that it can provide an adequate quality of service to a task. The meaning of quality of service is dependent on the application. The term includes: whether a maximum bound can be placed on the time a job will have to wait before starting execution; the minimum time quanta that a given job will be able to execute without interruption; and, the relative speed of a processor when compared to others in the processing pool. If the processor is assigned too many tasks, it may invoke a *transfer policy* to decide *whether* to transfer some tasks, and *which* tasks to transfer. A *location policy* determines which processor(s) will receive the tasks. There are two important models for location policies: sender-initiated and receiver-initiated. These depend on whether the sender polls potential targets for task transfer, or whether processors that are willing to receive extra tasks advertise the fact, respectively.

The advantage of dynamic load balancing over static scheduling is that the system need not be aware of the run-time behaviour of the application before execution. Dynamic load balancing is particularly useful in a system where the primary performance goal is the maximising of processor utilizations as opposed to the minimisation of runtime for individual jobs [207, 208]. This is often found in systems consisting of networks of workstations.

The main consideration involved in mapping threads to PEs is the requirement to balance the loads. It has been shown that load balancing is crucial to achieving adequate performance [53, 209]. Load balancing can be performed in one of two fundamental ways, where the decision of where to place the new thread is either made by a local queue or global queue. In a local queue, each PE makes its own decision; in a global queue there is a single point from where the threads are dispatched.

There are a number of methods that have been proposed for mapping a thread to a processor (from Feitelson [62] section 4.1.1) using local queues, for example:

- choose a PE at random [53];
- choose a PE at random, then assign the thread to its least-loaded neighbour (in the case of a non-switched network topology); and,
- probe a limited number of nodes and choose the least-loaded one.

Global queues are easy to implement in shared memory machines; they are not realistic options for distributed memory machines due to the need to maintain coherent lists of available tasks [62]. In most systems that use a global queue, tasks are allocated to a PE, execute for a time quantum, and are returned to the queue for re-allocation. The main advantage of using a global queue is that of load sharing, in which all processors get a roughly equal proportion of the workload. The disadvantages are the contention for the shared global queue, and the lack of memory locality.

3.1.3 Hybrid Static-Dynamic Scheduling

In the dynamic model, no placement information is assigned to the job. Consider the case in which a computation can be divided into a number of dependent parts. Initially, a node advertising the job's first service will be passed the job and once the task is complete, the node will consult its own records to determine the location (and cost information) of a server that can perform the next task, and will dispatch the job to that host.

This approach forces all hosts to be aware of, if not all, then at least a large part of the system. It does, however, provide for the truly dynamic nature of the system, in which nodes and services may or may not be available, and communication links may be down. This suggests that there may be problems with the dissemination of

server (and service) information to the remainder of the system. Problems may arise if the network becomes segmented.

The server estimates the cost of each part of the job's processing pipeline, and assigns each part to the processors in a manner that minimises the total execution time. Thus, the assignment of computation parts to processors is a static assignment. Problems arise if, after assignment, one of the servers selected to participate in the computation is aware of an alternative server that can perform the part at less expense. If the original schedule is used for execution, the computation will take longer than it could have through using the alternate server.

The onus, therefore, is on the current server to find the next, most appropriate server to continue the computation. It may be advisable to configure the system with facilities to 'backtrack' servers so if a server does not know about any appropriate servers to continue the computation, the job may be passed back to the last server to host it. The system would have to be carefully monitored, and trained not to follow local cost minima, which allow the computation to be passed to more remote, but more communications-costly nodes when it comes to return the results to the user.

This model runs into difficulties where the computation forks into a number of concurrent processing stages. One possible solution to this problem is to require that concurrent computations send periodic updates to the server which was used before the concurrent execution was started, to synchronise and continue the computation.

There needs to be some semblance of a priority system. To prevent users from abusing the priority system by requesting everything be run at the highest priority, it may be decided to take the solution that has been adopted in CCS [193], where the user is charged for the priority level at which they submit their job.

3.2 Independent Tasks

The performance of dynamic load balancing models in a heterogeneous system was studied by Chow and Kohler [42], where they considered deterministic and non-deterministic routing strategies. The non-deterministic strategy uses a probability, p_i of a process being routed to processor i, which is either chosen arbitrarily or based on a function of the system parameters.

Deterministic routing involves a job dispatcher which routes jobs to processors according to some policy. Three deterministic policies are investigated: minimum response time, minimum system time, and maximum throughput. Chow and Kohler's

study concludes that the maximum throughput policy, which uses information on the arrival rate as well as service rate and system state, is more optimal than those policies that base their decisions on the service rate and system state only.

We discuss the topic of independent tasks further in chapter 4. A tool is presented that allows the comparison of different static and dynamic scheduling algorithms for the placement of independent jobs. We use a number of simple placement algorithms and also a variant of Chow and Kohler's algorithm.

3.3 Dependent Tasks

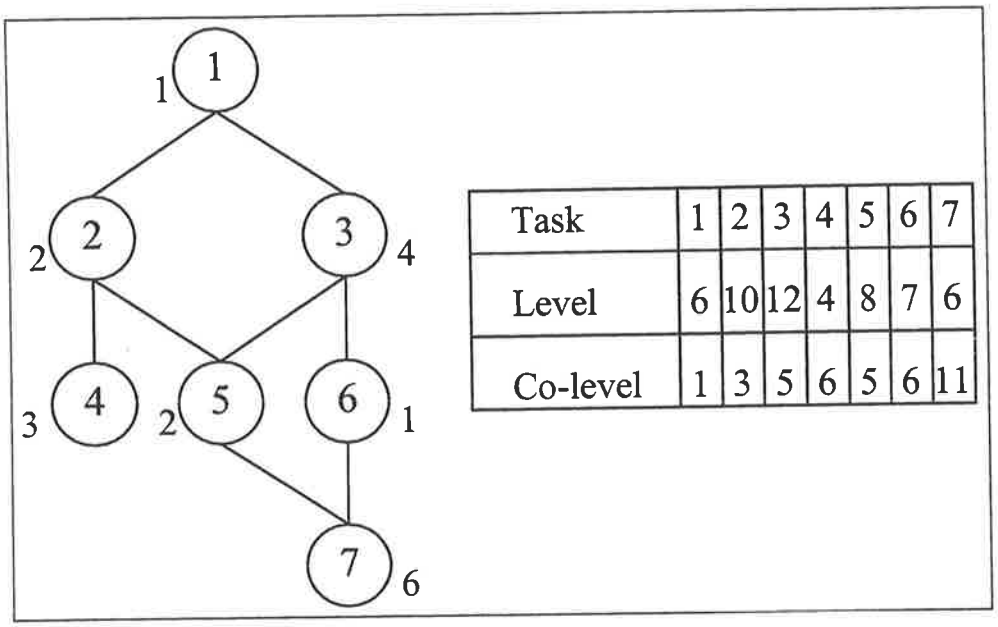
There are a number of different ways in which schedules for the execution of related tasks can be generated in the situation where the resources that are to be used are dedicated. Feitelson and Rudolph [63] recognise a number of different static approaches to parallel job scheduling, including critical path methods, list scheduling methods, and the partitioning of a DAG into clusters of nodes. As creating an optimal schedule is NP-complete, heuristic algorithms based on list schedules and clustering techniques are common [4, 86, 119].

3.3.1 List Schedules

List scheduling is relatively straightforward to implement. Most commonly, list scheduling is used to place the tasks that comprise a parallel program [4]. Tasks comprising a program are assigned priorities and placed in a list ordered in decreasing priority. Whenever tasks contend for processors, the highest priority task that is immediately executable is assigned first. List schedules may be preemptive or non-preemptive; if there are two tasks with the same priority that can execute, one is chosen randomly.

List scheduling algorithms rely on the knowledge of task execution times, and precedence relationships. They are most often used to schedule dependent tasks on clusters of homogeneous processors [4]. Given a set of tasks $T = T_1, T_2, ..., T_n$, which has the precedence graph G, we let the execution time of task T_i be t_i. The length of a directed path is defined as the sum of all the weights along the path including the initial and final vertex. The *level* of a task within a program is defined as the length from a vertex T_j to an exit node (a node which has no successors). Similarly, the *co-level* of a task is defined as the maximum distance from the vertex T_j to the entry vertex (which has no predecessors). A table can therefore be created in which each

task is assigned a level and co-level. Figure 10 shows a precedence graph and a table detailing each task's level and co-level. The level and co-level are used to order the tasks for execution. There is no allowance made for tasks not to be able to execute on any processor in the pool.



Task	1	2	3	4	5	6	7
Level	6	10	12	4	8	7	6
Co-level	1	3	5	6	5	6	11

Figure 10: List scheduling algorithms rely on task execution times and the precedence relations between tasks. A precedence graph is shown, with the task's level and co-level. The task number is represented by a vertex; the task's execution time is shown next to the vertex. The level of a task is the longest distance between the task and an exit node (which has no successors). The co-level is the longest distance between a task and an entry node (which has no predecessors). Tasks can be ordered by either the level or the co-level when being assigned to processors.

The trade-off between cost and accuracy of list schedules is discussed by Adam, Chandy and Dickson [4], where they consider the problem of scheduling two or more homogeneous processors to minimise the execution time of a program which consists of partially-ordered tasks. Independent tasks are not considered. They found that when the tasks that comprise a program are organised into a precedence graph, the most optimal schedules are found when using the levels of tasks, incorporating the task execution times to increase the estimate of the priority which to assign the task. In this case, the generated list schedules were found to be within 4.4% of the optimal execution time. Communications time between tasks was not considered in this study.

3.3.2 Clustering Algorithms

Clustering, or processor assignment, involves the collection of tasks that exchange a large amount of data onto the same processors, while at the same time distributing the tasks in order to achieve good load balancing [183]. Heuristics have been suggested for clustering [86].

Yang and Gerasoulis' dominant sequence clustering (DSC) algorithm [240] is one part of a multi-part graph scheduling system [84]. The algorithm clusters the dominant sequence of a directed acyclic graph (DAG), which is the critical path of a DAG whose nodes have been allocated to processors. The critical path of a DAG is the path connecting the programs which dominate the execution time of the DAG. If the programs on the critical path are not optimally scheduled, the resulting execution time for the job will be non-minimal. Once the dominant sequence of the DAG has been clustered, the remainder of the DAG is placed so as to minimise total execution time. The work on DSC was later applied to iterative task graphs [77, 239, 241].

An example of clustering is shown in figure 11. In this figure, nodes 1, 2 and 4 have been clustered together and assigned to host 1; nodes 3 and 5 have been clustered and assigned to host 2; and nodes 6 and 7 have been clustered and assigned to host 3. The decision of which nodes to cluster together are based on heuristics, including the amount of data to be shared between the nodes, and the relative speeds of each of the hosts. Clustering is discussed in the context of our model in chapter 5.2.3. We use an algorithm based loosely on DSC.

3.4 System State Information

The situation in which everything is known about the nodes participating in a distributed system is called complete system state information. This concept is shown in figure 12, where each node is aware of all operations that each of the others can support. In the case in which all nodes are under dedicated control of a master node, complete system state is not an unreasonable assumption. In a real distributed system, characteristics including the load on each machine, and the number of tasks awaiting execution comprise the reported system information.

In practice, the use of complete system state information means that the algorithm performing task scheduling does not need to estimate any of the characteristics of remote nodes. This allows a scheduler to make accurate predictions on the behaviour of a remote node. Most resource management systems (see section 2.1) assume

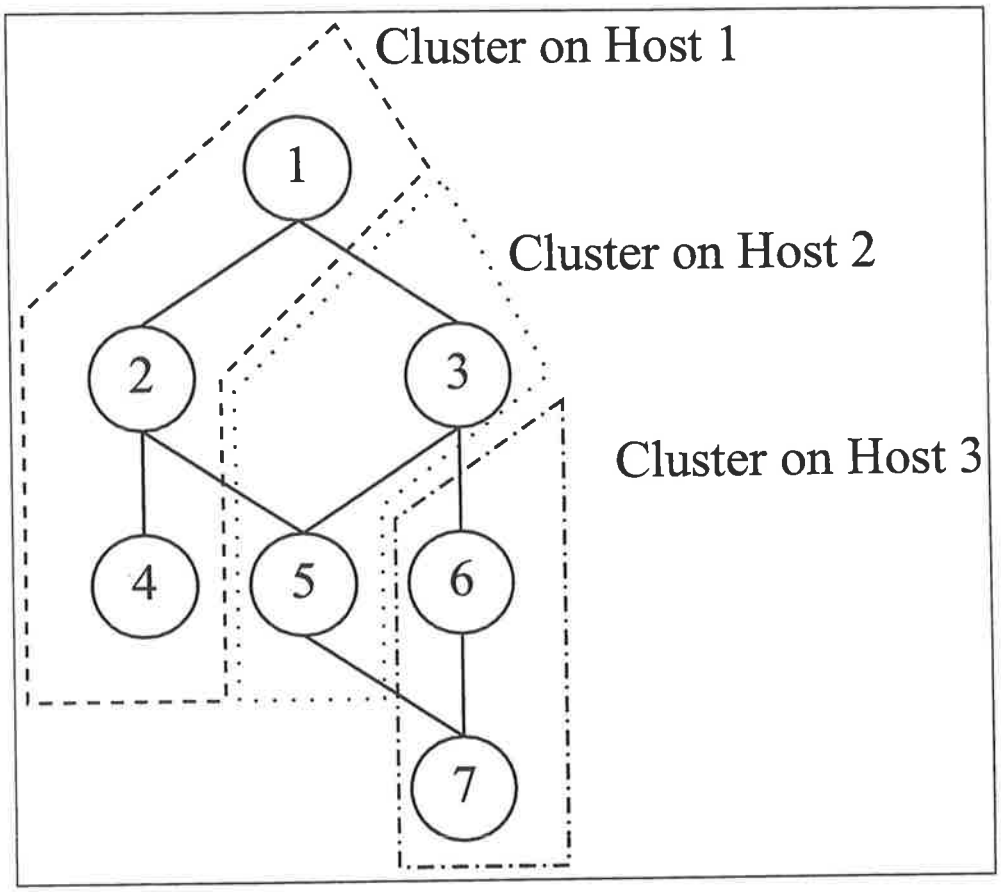


Figure 11: The nodes of a task graph may be clustered together to minimise the amount of data to be transferred between hosts in a distributed system.

they have complete system information, which is updated either by the master explicitly polling the slave machines; the slave machines regularly reporting their state information to the master; or as found in [172], there may be a hierarchical arrangement through which complete information is propagated and can be searched.

If the nodes are regular in behaviour and use, maintaining complete system state information may not be difficult. In the case where nodes are dynamically joining and leaving the distributed system, or are not dedicated to the use of the distributed environment, such maintenance is a non-trivial problem. In the case where maintaining complete system state information is too difficult or impractical, partial system state information may be still be of value.

Partial system state information allows nodes to keep their own view of the complete system state. This idea is illustrated in figure 13. Figure 13 i) shows

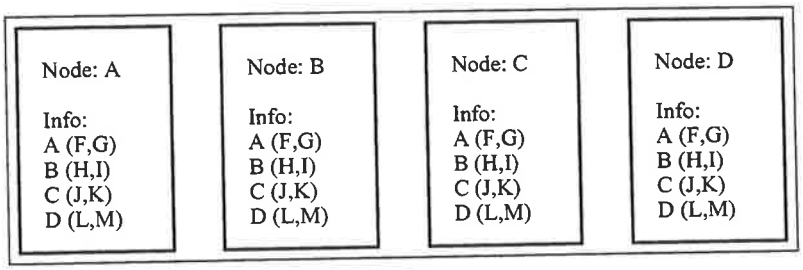


Figure 12: Complete system state information. Each node has complete, accurate information about all other nodes. For example: node A can support operations F and G; node B can support operations H and I; node C can support operations J and K; and, node D can support operations L and M. This information is replicated across all nodes in the distributed system.

the initial state of the distributed system where each node is unaware of the other nodes. Through some mechanism, nodes B and C may need to communicate. The results of this communication are shown in figure 13 ii). This may occur as: a product of a multi-cast request by either node; a manager at one of the nodes may be aware that the other is now online; or, after restoring some saved state information, either node may see some information that references the other, and seek to update its information.

The problem of partial system state information is similar to that of assuring consistency amongst distributed databases. An example of this problem is that found in the Internet Domain Name System (DNS) [160, 161]. Machines connected to the Internet always have a unique address, called the IP address [221], which is used for all communications. In addition, they usually have a human-readable names. For example, the host named `opal.cs.adelaide.edu.au` has the IP address 129.127.8.80. Within DNS, each logical group of computers is termed a *domain*. Domains can be split into further *sub-domains*, and many domains can be part of a larger domain in a tree-like fashion [9]. The information describing the mapping is stored locally to a domain. DNS exists as a mechanism for the exchange of local information between distributed domains. It relies on a central authority to delegate the authority to create domain names, and because of its hierarchical nature, when a hostname is unable to be resolved by the local domain, the request is passed to the larger, encapsulating, domain [199]. The design of DNS is not a directly useful approach to the problem we consider in the remainder of this thesis, as we do not impose a hierarchical ordering

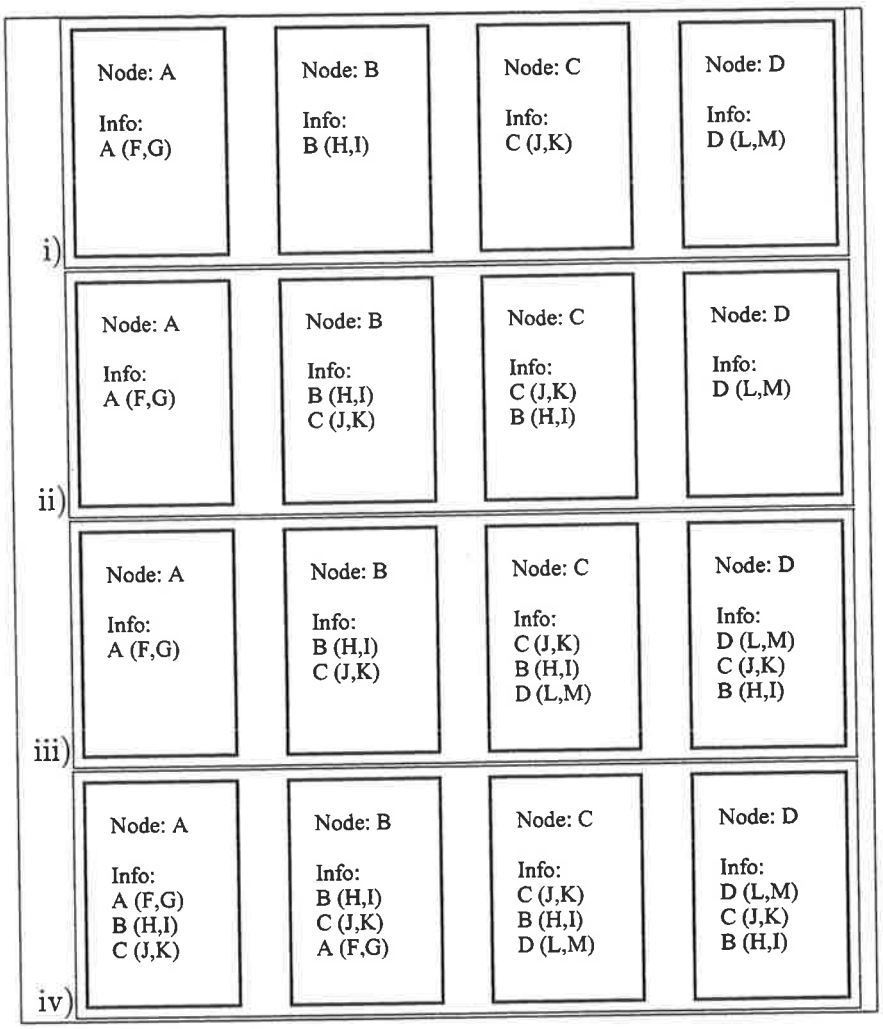


Figure 13: Consequences of partial system state information, When the state information maintained by a distributed system is not complete, some nodes may know about resources that others do not. i) Initially, each node only knows about itself and the operations it supports; ii) nodes B and C exchange knowledge; iii) nodes C and D exchange knowledge; iv) nodes A and B exchange knowledge. These exchanges result in partial system state knowledge being maintained about the system. If node A needs to know where an operation, L, is supported, then it can ask those nodes that it knows about, in the hope that they know of such a node. In the case of static information, complete system state information is attained quickly; the dynamic case is more complicated as information needs to be updated periodically.

of logical domains in the DISCWorld.

Although not as accurate as complete system state information, partial information is easier to maintain. Schedulers can use partial system state information when constructing placement schedules. We impose the restriction of partial system state information in the model and algorithm developed in chapter 5.

Static scheduling is simpler to implement and it seems to be the more widely-used approach to scheduling. The literature features many examples of static scheduling, where the authors have made different assumptions about such details as:

- the number of processors that are available for allocation [4, 28, 215];
- whether the processors can run any of the tasks that make up the application [42, 202, 224];
- the model used to evaluate the scheduling efficiency, and the input to the model [30, 41, 138];
- whether all tasks in the application have constant execution time [4, 41];
- whether communication time between the processors is taken into account [4, 28, 47];
- whether the model incorporates parallel, concurrent execution of tasks or both [41, 154, 224];
- whether loops or conditionals are allowed in the input to the model [202, 224]; and,
- whether the processors that make up the working set are homogeneous or heterogeneous [30, 138, 215].

3.5 Historical Review of Scheduling

This section presents an historical review of scheduling literature. Of course, this is not an exhaustive list of all the literature; it encompasses what we believe to be important studies. We are not interested in the complexity of the models – just the models themselves.

Table 3 shows a comparison of models found in the scheduling literature. The majority of the systems studied were formulated in the context of parallel program

Authors	Ref	Sched Type	Task Alloc	Model Inputs	Task Exec	Comm Time	Type of Procs
Stone	[215]	H	S	Arb Graph	K	V	Hetero
Kaufman	[138]	S	U	Tree	K	N	Homo
Adam, Chandy, Dickson	[4]	S	U	Prec Graph	U	N	Homo
Chow, Kohler	[42]	S	U	Unrel tasks	K	N	Hetero
Bokhari	[28]	H	U	Tree	K	V	Hetero
Chou, Abraham	[41]	S	U	Arb Tsk Gr	K	Y	Hetero
Towsley	[224]	S	R/U	Arb Tsk Gr	K	Y	Hetero
Eager, Lazowska, Zahorjan	[53]	D	U	Unrel tasks	U	Y	Homo
Sarkar, Hennessy	[202]	S	U	Prog funct tree	K	Y	Homo
Cvetanovic	[47]	S	U	Prec graph	K	Y	Homo
Shen, Tsai	[204]	S	U	Prog func tree	K	Y	Hetero
Bokhari	[29]	S	U	Prog graph	K	N	Homo
Polychronopoulos, Kuck	[189]	D	U	Tree	K	N	Homo
Bokhari	[30]	S	U	Tree	K	N	Homo
Litzkow, Livny, Mutka	[153]	D	U	Unrel tasks	U	N	Homo
Kruatrachue, Lewis	[143]	S	U	DAG	K	Y	Homo
Pulidas, Towsley, Stanovic	[190]	D	U	Unrel tasks	K	Y	Homo
Lo	[154]	S	U	Arb task graph	K	Y	Homo
Fernandez	[65]	S	U	Tree	K	Y	Homo

Table 3: Comparison of models found in the scheduling literature. The type of scheduling can be either Static, Dynamic or Hybrid (static then dynamic). Task allocation can either be Restricted, Semi-Restricted or Unrestricted. The model inputs can be an Arbitrary Graph, a Tree, a Precedence Graph, a DAG, or Unrelated Tasks. Task execution time is either Known or Unknown, and inter-task communication time in the model is either included as a constant constant (Y), included but variable (V), or not included (N). The processors used by the model are either homogeneous or heterogeneous.

execution on tightly-clustered homogeneous processors. It can be seen that most of the systems do not incorporate the concept of restricted program placement between nodes – they assume that all programs can be placed on any processor in the pool. In addition, the input to most of the models is a tree, or graph, of programs (or tasks). Tasks organized into a tree have precedence relations; they are instances of task graphs with multiple source or sink nodes. In most cases, an arbitrary task graph, or a precedence graph, is implicitly directed. If the directed graph does not contain cycles, it is termed a directed acyclic graph (DAG). DAGs are used to represent processing requests in the model and algorithm developed in chapter 5. Some of the systems featured in table 3 are described in the remainder of this section.

Stone [215] used a modified commodity flow algorithm and cutsets for scheduling an arbitrary graph over two and three heterogeneous processors. Each task had known, but not identical execution time. Each branch in the network had an associated amount of information to be transferred. The capacity of the network, was used to measure the actual amount of information flowing between the tasks at runtime. Cutsets divide the graph into information sources and sinks. The model incorporated variable communication time between processors. It also allowed concurrent execution, but neither parallel execution or loops and conditionals in the input graph. The model had the main characteristic of the input task graph being undirected, thus representing two-way communication between modules; the cost of moving a computation to another processor is traded off against communication costs of transferring the results between processors. This model was one of the first to consider semi-restricted tasks. Some tasks were assigned to processors; others were able to 'float' between processors during program execution.

Kaufman [138] used a longest path method to statically schedule a tree of tasks onto n homogeneous processors, in an unrestricted fashion. The longest path method is equivalent to a list scheduling algorithm, ordering by decreasing task level. Trees of tasks have precedence relations between tasks. Tasks were assumed to have known but non-identical execution times, but communication time was not incorporated into the cost model. Concurrent execution of tasks was supported, but not parallel execution, or loops and conditionals in the input tree. This was the first study to consider tasks with non-homogeneous execution times.

List schedule algorithms were compared in [4], across n, where $(n \geq 2)$ homogeneous processors, using a precedence graph as input. Tasks were assumed to be unrestricted in assignment, and had random, but deterministic execution times.

Communication time was not incorporated into the cost model. Concurrent execution time was allowed, but not parallel execution, and loops and conditionals were not supported. If tasks are treated as programs, and task intercommunication is ignored (i.e. tasks are independent), list scheduling is equivalent to the first-come first-serve scheduling algorithm used in the tool described in chapter 4.

Bokhari [28] featured a static then dynamic scheduling algorithm, using unrestricted placement of a tree of tasks, across $n, (n \geq 4)$ heterogeneous processors. Tasks had known but non-identical execution time, and variable communication time was incorporated into the cost model. This study was one of the first to consider variable communication times. Parallelism between tasks, loops and conditionals in the input tree were not considered. All processors are assumed to be dissimilar, but are able to execute any task in the tree. The shortest tree algorithm is used to minimise the sum of execution time and inter-processor communication time. The algorithm suggested by this study exhaustively iterates through all possible assignments of modules to processors, using the shortest tree. This algorithm is similar to the model that we develop in chapter 5.

Chou and Abraham [41] suggested seven program descriptors for dynamic execution on distributed systems and policy iteration. The descriptors are: execution time of a task; the communication time for the results of a task; probability a task fails on a processor; the time to create a checkpoint for a task; time to restart a failed task; time to initiate a set of concurrent tasks on a processor; and, the communication time for the results of a set of concurrent tasks. The paper presents an algorithm for optimal task assignment on n heterogeneous processors, featuring probabilistic conditional branches in the arbitrary input task graph. The study was one of the first to consider parallel as well as concurrent execution of application tasks, and although communication was considered in the cost model, the communication cost between processors was fixed.

Towsley [224] extended Bokhari's shortest path method, and incorporated the concept of processor reliability from [41] to produce run-time estimations by loop-unravelling. An arbitrary task graph is scheduled for execution on n heterogeneous processors, where tasks are either allowed to execute on any processor, or they are all assigned to certain processors. Task execution time is known but non-identical, and communications time is incorporated into the cost model, albeit fixed. This research is significant because it supports conditionals and loops within the arbitrary task graph as input.

Chen and Eshaghian [40] present a fast recursive mapping algorithm for parallel applications on parallel systems. Their system involves the clustering of the algorithm's task graph, and the separate clustering of the parallel system's graph. The algorithm provides for the mapping between the two clustered graphs in $O(MN)$ time, where M is the number of task modules and N is the number of underlying processors. In this study, the authors assume that each module (comprising the application) executes for a single time unit, and communicates a uniform amount of data to any child modules. They further assume that each processor is the same speed and that all have the same network characteristics (transmission rate and latency). Although this study is not applicable in the situation we consider for this thesis, it is applicable to the case where only a single variable (either the program graph or the architecture of the parallel machine) is changing. Thus, many program tasks, for example, could be mapped to a single parallel architecture graph.

The automatic heterogeneous supercomputing (AHS) system [49] uses a semi-dynamic strategy for scheduling user application programs. The system maintains a file for each application program containing an estimate of the execution time of that application on each of the available machines; when the user invokes the program, a machine is chosen such as to minimise the turnaround time, which is a function of the current load on the machine and the expected execution time on that machine. The system is semi-dynamic in that once the application is started on a given machine, there is no intervention from the scheduler. The amount of information that this scheduling algorithm uses is minimal, and while it benefits from low scheduling overhead, it does fail to take into account any issues arising from the input data locality.

The self-adjusting dynamic scheduling (SADS) algorithm [101] develops a detailed cost model which is used to trade-off effects of processor load imbalance, interprocessor communication delays and scheduling and synchronisation overhead. It creates partial schedules, minimising the cost function by using an algorithm similar to branch-and-bound [236]. The time that is taken to create the partial schedules is bound by the speed with which least-loaded processor in the pool can complete its tasks. Once the least-loaded processor completes its tasks, the computed partial schedules are send to all the machines in the pool, and the process begins again. This algorithm suffers from the need to have a dedicated processor to create the partial schedules, whether this processor is the same all the time, or whether it is chosen at execution time. This is extended in the self-adjusting scheduling for heterogeneous systems (SASH)

algorithm [102]. The SADS algorithm is similar to the continual adaptive scheduling algorithm for independent programs that is considered in chapter 4.

The scheduling scheme suggested by **Shirazi, Chen and Marquis** [205] involves the clustering of a program's task graph into linear tasks, and then statically scheduling them after optimisation. Linear clustering of nodes is used to decrease the effects of inter-processor communications, while parallelising the nodes reduces the execution time. The implementation of this algorithm has the drawback that one of the assumptions that it makes is that the task graph that is to be used as input has a single root node, that an unbounded number of target processors are available, and they are homogeneous. No consideration is given to contention that may arise from running multiple programs at once, or the effects of introducing heterogeneity into the processor pool.

Weissman and Grimshaw [233] present a framework for partitioning data parallel computations in heterogeneous environments, which is implemented in Mentat. This framework is designed to effectively utilise clusters of workstations and supercomputers with different communications topologies, with a view to reducing the total elapsed time as seen by a program. Different communication topologies are characterised by different cost functions. This study includes the notion of router delays and per-byte data coercion (between data formats) costs. The values used for communications costs in the study are for an ideal system in stand-alone mode, and do not consider the effects of real-time machine load. The process placement algorithms used are communication topology-dependent, and rely on callbacks from the tasks whilst they are in computation and communication phases to provide the information that is needed to make intelligent placement decisions.

Shirazi, Wang and Pathak [206] consider the effects of three static scheduling algorithms in the context of a multiprocessor environment, where the execution time of each of the components of the input are known and invariant. Their study concentrates on achieving the fastest turn-around time for an input DAG. The algorithms they consider are the critical path method, and two novel algorithms, a heaviest-node first method, where the longest-executing program nodes are assigned to the least-loaded processors first, and a weighted length algorithm which extends the critical path algorithm by incorporating such information as the branching factor of each node, the number of children, and their weights. Although novel, and effective in terms of computational complexity, the models do not take into consideration the delay effects of interprocessor communication or the effects of any processor

heterogeneity.

El-Rewini and Lewis consider the problem of scheduling parallel program tasks onto arbitrary machines [57] in the presence of contention between processing nodes. They use a scheduling heuristic, MH, which produces a static schedule based on list scheduling. A task graph and a undirected graph, representing the target machine, are used by the mapper algorithm, which produce a Gantt chart of the resulting schedule. Contention on interconnection links is modeled as a extra delay which is constant for a given link. Although the effects of contention on interconnection links is ignored, we consider the effects of assigning multiple tasks to a single processor in the model we develop in chapter 5.

Malloy, Lloyd and Soffa [156] extended existing research based on the critical path method to the fine-grained parallelism found in program instruction streams, and implemented their model on a two-processor homogeneous system. This research is too fine-grained to be applicable to the problem that is under consideration.

3.6 Discussion

While a variety of different scheduling and placement models have been studied in the literature, there is very little evidence of most of the models being used in practice. Although scheduling and process placement are vital parts of a distributed system, most of the cluster computing environments described in chapter 2 do not seem to have been designed with any algorithms in mind.

There is a great deal of existing research that focuses on scheduling DAGs across a bounded or unbounded number of target nodes. Although useful, results which only consider a homogeneous collection of nodes, such as [4,138,203], are not helpful in the case of heterogeneous nodes. Still more research [156] focuses on exploiting fine-grained parallelism found in instruction streams. That too, is not helpful in the case where a processing pool is comprised of heterogeneous nodes separated by a wide-area broadband network.

Unfortunately the existing research (eg [4, 56, 57, 63, 84, 85, 156, 201, 242]) that considers a collection of heterogeneous nodes all assume that either:

- components can run on every machine in the processor pool; or
- interconnection networks are all homogeneous; or

- nodes are dedicated and are always available to accept a new component for execution; or
- the size of the data to be transferred is the same; or
- the nodes do not fail; or
- a centralised scheduler is acting in isolation and is in possession of up-to-date global state information; or
- all the data needed by a program component is available from the node on which it runs.

The general case that we consider has the following characteristics, for which an adequate solution does not yet exist:

- program components, or services, are not available on every node;
- interconnection networks vary between nodes;
- resources need to be scheduled across administrative boundaries;
- nodes are owned and maintained by separate groups (and individuals) who may arbitrarily decide to remove the machine from the available processor pool;
- the data outputs of different services are not the same size, nor are they necessarily constant across executions of the same service with different input parameters;
- execution schedules can be created by any node within the processor pool, using the most up-to-date information available, with each node creating its own schedules; and,
- in some situations, the data that a service requires may not be readily copied to another machine, which may necessitate moving the service code to the remote machine.

3.7 Conclusion

The scheduling literature provides solutions for a number of special cases. However, we believe that the previous results are not completely applicable for the system model

that is found in some wide-area distributed systems, such as our DISCWorld prototype metacomputing environment. They do not address the fundamental constraints and features found in our system: lack of up-to-date global system information; restricted placement and mobility of data and services; global naming strategy for data and services, by which data may be re-used; and the single assignment nature of data.

There are a number of popular approaches to this problem, most of which are related to the critical path method. We consider services that can be executed on a number of different nodes within the distributed environment, each with a different set of transfer and execution costs. Due to the sheer computational complexity of computing a critical path for every possible node assignment, we feel the classical approaches of list scheduling [4] and clustering [201, 240] are not applicable. In addition, the above techniques fail to take into account the probable need to transfer input data to each service before it can be run – which is different from the data created as the output of a previous processing step. Finally, Sarkar's two-stage algorithm [201] and DSC only consider the case of homogeneous compute nodes, which is too restrictive for the more general case we consider.

A number of heuristics and partial solutions may be combined to form a solution to the problem we consider. We use the general idea of clustering, and in particular, a modified version of Yang and Gerasoulis' DSC algorithm to minimise the total execution time of an individual job.

We consider a processing request to be made up of a number of programs, which may have inter-dependencies. When all processors in a distributed system are controlled, and assigned programs by a single front-end processor, static scheduling, as described in section 3.1, is best. This is because the complete state of the system is knowable. There are two major drawbacks to using a centralised scheduler. Firstly, the use of a centralised scheduler may cause a processing bottleneck, and secondly, the centralised scheduler introduces a single point of failure. This means that if the node hosting the scheduler suffers a failure, either of the physical machine or the network interconnecting the scheduler with the remainder of the processors in the pool, no more jobs will be able to be scheduled.

If all processors have the ability to assign programs to run in the distributed system, then dynamic scheduling, or load balancing, is more appropriate because of the lack of control by an individual node over the whole system. Most dynamic scheduling algorithms seek to obtain the best performance for individual programs under consideration, in the assumption that by doing so, the complete processing

request will be efficiently executed; they perform no forward-planning.

In chapter 4 we present a tool we have developed that allows quantifiable analysis of static and dynamic scheduling algorithms when used to execute independent programs injected to a cluster of computers. When the distributed environment is relatively stable, we find that in the case where the programs to be run are easily characterised, a static scheduling algorithm provides good execution time; when the programs are not easily characterised, a dynamic algorithm is needed.

In the situation where the node from which a processing request is submitted has information on the system state (even if incomplete), there is benefit in using that information to produce a heuristically good execution schedule. If the execution schedule is created with static task-to-processor assignment, and then modified at execution time, we call this model hybrid static-dynamic. Thus, the node creates the best static schedule that is heuristically possible using its current system state information as an estimate. Since other nodes may have different system state information, at execution time they may modify the schedule to make it more efficient for the processing request's execution.

Some of the systems we review in chapter 2 provide solutions which incorporate subsets of the conditions which characterise our system. In chapter 5 we develop a model for scheduling in wide-area systems. The DISCWorld model is designed for jobs that consist of inter-dependent high-level programs, which we subsequently describe in chapter 6.

Chapter 4

Scheduling Independent Tasks

4.1 Introduction

Optimal scheduling of interacting jobs on metacomputer systems is a difficult problem. Here we focus on the simpler problem of scheduling non-interactive, independent tasks. This work is also reported in [131]. Independent tasks are described in chapter 3.

In implementing metacomputing environments it may be infeasible or impossible to have a pre-set up point-of-presence, or daemon, running on each node in the system. The node may not be able to support the environment in which the daemon is written. For example, Java [88] is not available for some platforms such as the Thinking Machines CM-5. The node's administrator or owner may not want the machine directly accessible to the remainder of the metacomputing environment. For example on a Beowulf cluster [115] a front-end node may be acting as a gateway to hidden worker nodes. It is therefore sensible to design a software scheduling system to allow a designated node to act as an active gateway for other passive nodes.

Under this model, the active node must be able to schedule remote processes to provide good job throughput in the larger scale of the whole metacomputing environment, which may comprise a cluster of compute clusters. It must also ensure that fair access is granted to resources which may be shared. Metacomputing resources often comprise individual workstations and other computers accessed by their owners and other users *directly*, outside the control of any scheduling software.

In this chapter we focus on the problems of scheduling on cluster computing. There is already a large body of existing work in this area, including research projects such as: NQS [117]; PBS [23]; DQS [220]; EASY [151]; and the Prospero Resource

Manager [172] (PRM), as well as commercial products such as: LoadLeveler [123]; and Codine [81]. These software systems are broadly characterised as queueing software which runs on *all* of the participating nodes, and are described more fully in chapter 3.

Although several support partial node availability they are often hard to set up and maintain with policy information spread across distributed nodes. These systems do however use individual node loads in load balance calculations, which is more difficult for a centralised scheduler that does not have access to daemons on each node. In this chapter we experiment with a centralised scheduler model to manage a sub-cluster of resources. A full metacomputing environment may integrate many such transient sub-clusters together using the decentralised daemon approach with each sub-cluster managed by a daemon. This is useful if transient sub-clusters comprise resources temporarily assigned from other duties such as individual users' workstations.

We consider a number of scheduling algorithms. We measure their behaviour when all the jobs that are to be run on a particular sub-cluster are registered *before* the scheduling process begins, but the load state of the system is unknown. While a job might take a relatively well defined time to execute when all required input data is available on the local node, this can increase significantly when it must retrieve data either from the local gateway or a remote system.

The typical usage scenario we are interested in for our prototype DISCWorld [113] metacomputing environment is for high level user queries to be decomposed into well defined independent jobs. These jobs may be parameterised simulations, or simple data filtering jobs. The execution model is that of a gateway node which hosts a scheduling daemon which can remotely execute jobs independently on the sub-cluster it is responsible for.

4.2 Scheduling Independent Jobs

In a set of independent jobs, one of the defining characteristics is the execution time of each job. In the situation where each job has similar resource requirements, it is possible to test the performance of a scheduling algorithm on different distributions of job execution times. In this study, we approximate the time taken to copy the data necessary for the job to be executed, and also the effects of different sized data by considering different distributions of execution time. We consider five different distributions of task execution time: homogeneous; bimodal; uniform random; normal; and Poisson. All results we report in this chapter were carried out on

a cluster of 8 DEC Alpha Workstation running Digital Unix. We are experimenting with the effects of heterogeneous cluster nodes but do not report on that work here. The use of a homogeneous cluster of workstations allows us to see the impact of the different scheduling policies of different job mixtures. The job execution times that we used for the different distributions are shown in table 4.

Distribution	Characteristics
Homogeneous	job duration = 0.1sec
Bimodal	job duration randomly chosen from 0.1 or 10sec
Uniform Random	job duration in interval [0.1,10)sec
Normal	mean = 0.1sec, σ = 1.0sec
Poisson	mean = 0.1sec, σ = 1.0sec

Table 4: Simulation job duration characteristics

As the name suggests, jobs with *homogeneous* execution times all run for the same duration. There are many examples of jobs with a homogeneous execution time distribution, most commonly found when requesting that the same application be run with either no inputs, slightly different, but equivalent inputs which are already present on the local node. *Bimodal* distributions occur when the jobs they represent take one of two durations to execute. In the context of a metacomputing environment, this distribution occurs most often when a single job is being executed which sometimes has to retrieve input data from a remote node. For the purposes of this study, when jobs have a bimodal execution time distribution, they have an equal (50%) chance of being randomly assigned to be a long or short job. *Uniform random* distributions must be used to model jobs when nothing is known about their execution time. An example of this is when a job has roughly equal probability of being able to either run without any parameters, or it may need parameters that can vary in size, and may be available from the local node or a remote node. A *normal* distribution occurs most frequently when a job is used with input data that is only loosely clustered around a mean size, which is most often available from the local node. An example of a Poisson job distribution is a job which is usually run with input data that is very tightly clustered around a mean size and which is nearly always resident on the local node.

4.3 Scheduling Algorithms

Efficient scheduling software should minimise idle processing time. No single algorithm can achieve optimal performance on all possible job execution time distributions. However, given extra information such as which distribution dominates the job spectrum a smart scheduling system can choose which policy will be near optimal. To investigate this, we tested five scheduling policy algorithms against some simple job time distributions. We implemented these algorithms in a prototype centralised scheduler program known as *dploader*. This multi-threaded program was implemented in C and uses a single controlling thread to assign jobs to remote processors and an additional single thread per processor executing remote jobs. The program structure is shown in figure 14. It therefore acts as the central sub-cluster manager discussed above and provides a test framework. At run time, the *dploader* program is set up with a host list of those remote nodes or hosts to which it is able to assign jobs.

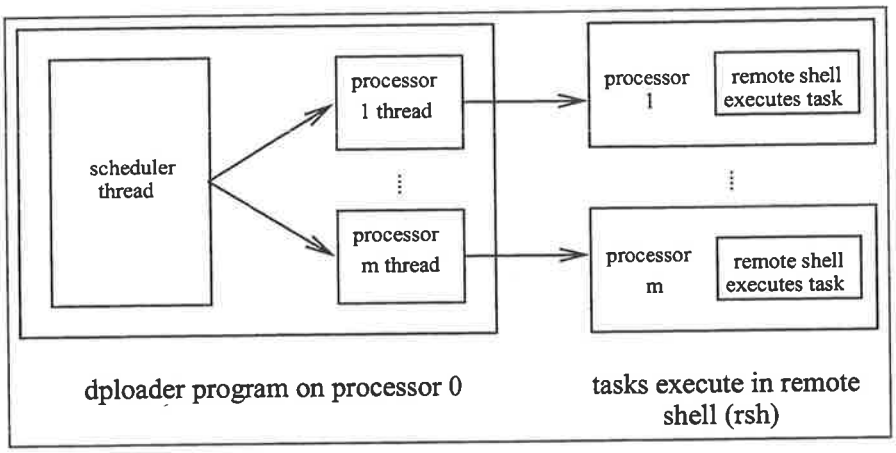


Figure 14: `dploader` program structure. The scheduler creates a thread per processor and a controlling thread. The per-processor threads use `rsh` to execute remote tasks.

A number of scheduling algorithms were tested, including: round-robin (perfect); round-robin with clustering (perfect-clustering); minimal adaptive (adaptive); continual adaptive (continual-adaptive); and first-come first-serve (fcfs). The names in parentheses are used as shorthand on the graphs and tables of results.

Scheduling is said to be *static* when the processors on which the jobs will run are assigned at compile-time or before execution. *Dynamic scheduling* [6] or *load balancing* is performed at run-time. This involves all jobs being assigned to processors, and the use of transfer policies and placement policies in order to decide when a job will be

moved between processors and to where it should be moved, respectively. This is further described in chapter 3.

Our problem is that we are not able to run daemons that report accurate and up-to-date sub-cluster node status information to the centralised scheduling software. The only information that we receive from a remote node is the execution time of the last job, or if multiple jobs were run, the average execution time. In our metacomputing context, the classic terminology of static and dynamic scheduling is no longer entirely appropriate due to its inability to describe run-time job assignment algorithms which use minimal information about remote nodes. These algorithms are not strictly static as job placement is performed at run-time. They are also not strictly load balancing algorithms in the sense that a centralised algorithm decides job placement and assignment until jobs are actually ready to be executed.

We define performance as the total time taken to schedule and execute the jobs assigned. This metric incorporates not only the time taken to execute jobs on the remote machine, but also overheads associated with the establishment of remote shells, and any other time spent waiting for a job to be assigned to a waiting processor. The time taken to establish a job (or group of jobs) on a remote machine is termed the job assignment latency. We assume that the additional increase in latency to request a reasonable number of jobs is negligible because the majority of overhead making up this latency is caused by contacting the machine and creating a remote shell or process.

In **perfect** or **round-robin** scheduling, each node receives an equal proportion of the jobs. Thus, if there are n jobs and m nodes in the workstation cluster, then each machine will receive $\frac{n}{m}$ jobs to process. Nodes are cyclically assigned a single job in the order of their appearance in the host list after the preceding host has been assigned; thus node $i+1$ will not receive a new job until it is possible to assign a job to node i. Processor 0 receives job 0, processor 1 receives job 1, ..., processor 0 receives job m, processor 1 receives job $m+1$ and so forth. This is illustrated in figure 15. It can be seen that this implementation of perfect scheduling may lead to node $i+1$ being idle if node i is relatively slower. This algorithm is best suited to the case in which the processors that comprise the sub-cluster are homogeneous, both with respect to node performance, and actual load on the processors. Perfect scheduling is designed to average out the number of jobs executed on each host. Unfortunately, this algorithm is expected to be the worst performing of those studied, for several reasons, including those of job and node non-homogeneity in practice, as well as high

individual job assignment latency.

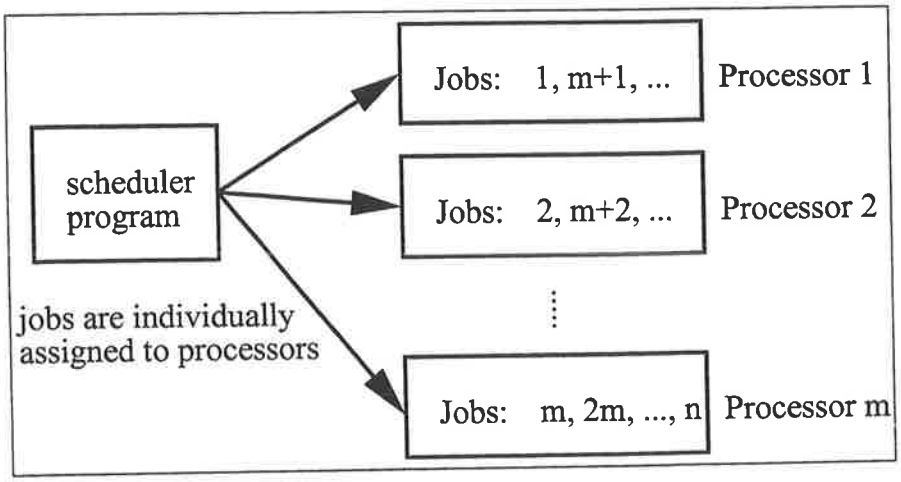


Figure 15: Round-robin scheduling of independent jobs. Processors are not assigned a job until the preceding processor has finished its job. This scheduling algorithm ensures a perfect proportion of the total jobs will be assigned to each processor.

In the **clustered round-robin** algorithm, the total number of jobs are divided amongst the host list. Therefore, a processor receives all of the jobs it must execute at startup. Thus, processor 0 receives jobs $0 \ldots \frac{n}{m} - 1$, processor 1 receives jobs $\frac{n}{m} \ldots \frac{2n}{m} - 1$, and so forth. This is shown in figure 16. This algorithm has the advantages of being the easiest out of those studied to implement, and also that it reduces the job assignment latency. However, it suffers from the deficiency of not being able to control the number of jobs to execute if a processor is found to be relatively slower than the others. In terms of job assignment latency, this scheduling algorithm is the least expensive of those studied, as it only performs a single job assignment on each remote processor. Thus the execution time of the whole program is that of the slowest processor, or if the processors are homogeneous, the processors that gets assigned the greatest number of longer jobs in a distribution.

The algorithm we term **minimal adaptive** scheduling is that of testing the relative performance of nodes *before* placing the remaining jobs onto them. The object of this is to avoid the Amdahl effect incurred by the clustered round-robin algorithm. Provided $n > m$, where n is the number of jobs and m is the number of processors, we select and place the first m jobs onto the nodes, one job per node. After timing how long it takes for each node to process these single jobs, a relative ranking of nodes is made, and the remaining $n - m$ jobs are proportionally assigned to the nodes, as shown in figure 17. This algorithm does not incur as much job assignment latency as the perfect scheduling algorithm, but as it performs two sets of job assignments (one

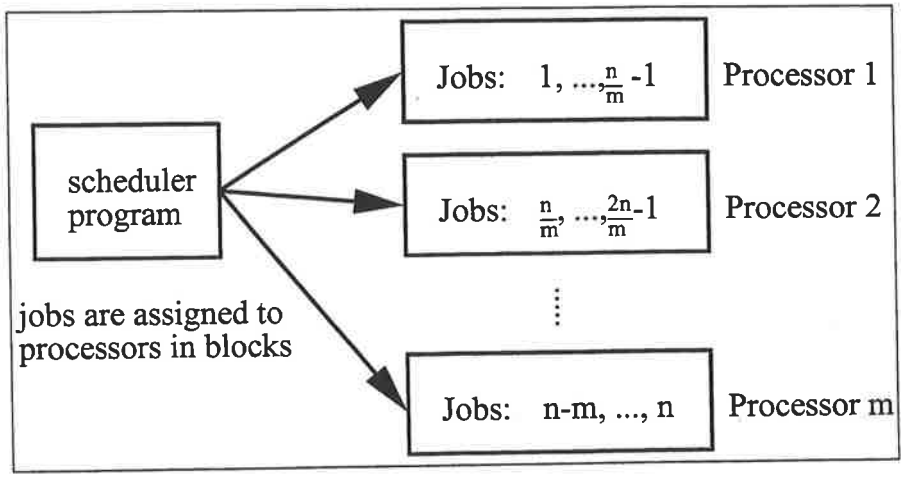


Figure 16: Round-robin scheduling of independent jobs with clustering. The total number of jobs is divided (as equally as possible) amongst the available processors.

to perform benchmarking on the remote hosts, and the other to assign the remainder of the jobs), it suffers from double the latency of clustered round-robin scheduling. Minimal adaptive scheduling makes the underlying assumption that the state of the target processor for the duration of the first job is indicative of the steady state of the machine. Furthermore, the algorithm ignores job heterogeneity. It is quite possible that if the first m jobs that are in the queue waiting to be processed are not the same, then the resulting estimate of the relative processing capacity of the target processors will be incorrect. Fluctuations in the machines' load are ignored by this algorithm.

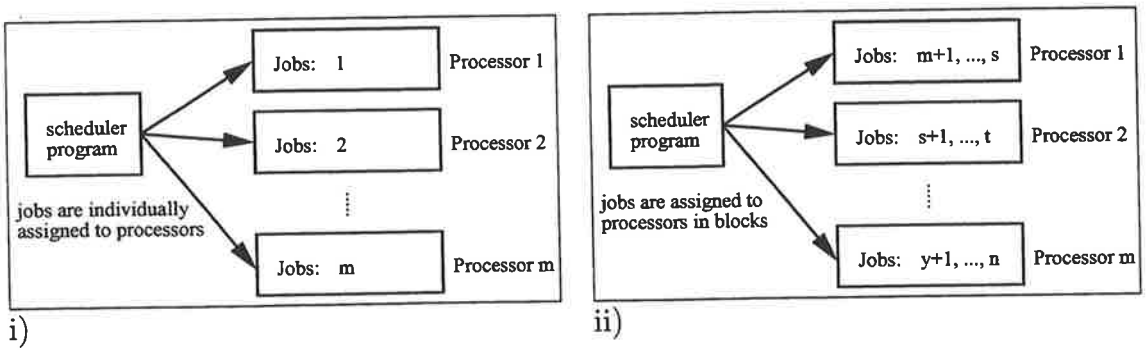


Figure 17: Minimal adaptive scheduling of independent jobs. In the first step, (i) a single job is assigned to each of the processors. The remainder of the jobs are proportionally assigned depending on their relative speeds (ii). (s, t, y depend on the relative speeds of the processors, and $s, t, y < n$.)

Continual adaptive scheduling is an algorithm designed to achieve the best

turnaround time for a group of jobs. Jobs are assigned to each processor in groups of t, a tunable parameter. After each processor has finished processing its group, the group execution time is recorded, and the total number of jobs that the processor will be able to execute such that all processors should finish at approximately the same time, is estimated. This concept is shown in figure 18. The continual adaptive scheduling algorithm performs better than the perfect algorithm, but not as well as clustered round-robin nor as well as the minimal adaptive algorithms in terms of job assignment latency. This is due to the need for the algorithm to assign blocks of jobs to the remote processors, and analyse the performance of each, before assigning new jobs.

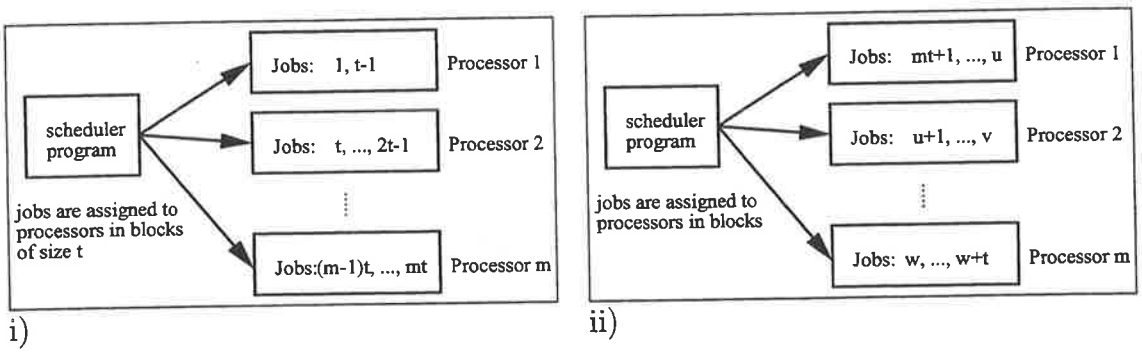


Figure 18: Continual adaptive scheduling of independent jobs. In the first step, (i) a block of t jobs is assigned to each of the processors. Based on the relative speed of each processor in processing the previous block, a maximum of t jobs are assigned to the fastest processors so that each processor will finish at the same time. (u, v, w depend on the relative speeds of the processors, and $u, v, w < n$.)

First-come first-serve (FCFS) scheduling refers to the technique of placing all available nodes into an ordered list, and assigning each job individually to the node at the front of the list. Once a node is assigned a job, the node is removed from the head of the list until it has finished processing its assigned job, when it is added to the end of the host list. This algorithm has the benefit of favouring the faster nodes, as the faster a node processes a job, the quicker it will be added on to the end of the list, and the sooner it will be assigned a new job. This is illustrated in figure 19. This scheduling algorithm provides the best throughput with respect to job execution, but will cause the machines to be used in an unbalanced fashion. The major deficiency of the algorithm is its need for a low job assignment latency, since jobs are assigned individually.

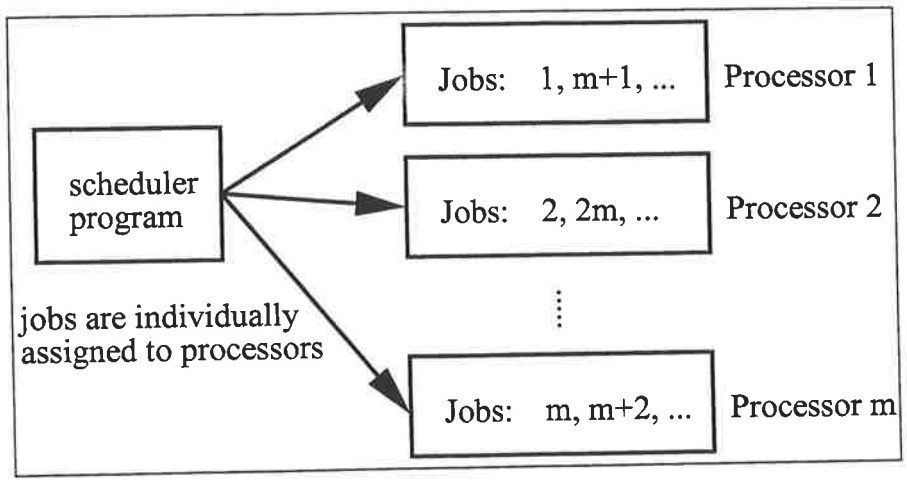


Figure 19: First-come first-serve scheduling of independent jobs. Each processor is assigned a job as soon as it finished its current job.

4.4 Algorithms Test Results

The five scheduling policy algorithms were tested on a variety of homogeneous and heterogeneous clusters of processing nodes, using many independent job distributions. Each distribution of between 1 and 1000 jobs was drawn from the stated statistical distribution. In the limit of $n \gg m$, where each machine had a large number of jobs, the scalable behaviour is summarised in table 4.4. This shows the job throughput and job assignment latency overhead figures, which scale linearly as expected, but which differentiate the efficiency of each algorithm for different job time distributions. Figure 20 shows the $n \approx m$ behaviour, which illustrates some of the characteristics of each algorithm.

Generally, the homogeneous distribution evokes the best performance from each of the considered scheduling algorithms. This is because of the uniformity in job execution time, and allows us to view the impact of job assignment latency.

The round-robin scheduling algorithm, shown in figure 20 i), performs no prediction at all. Hence the performance of this algorithm is reflected in a distribution's clustering around the mean job execution time in a synthetic job load. Thus, the jobs exhibiting a homogeneous execution time distribution perform best, followed by Poisson, normal, uniform random and then bimodal. The high variability in the Poisson, normal, uniform random and bimodal distributions causes delays in the assignment of jobs to processors. Because jobs are individually assigned to processors, a high component of the overall performance is the job assignment latency.

On average the slopes representing job execution time distributions in clustered

Scheduling Algorithm	Job Execution Time Distribution	Execution Time/Job(s)	Job Assignment Latency(s)
Round Robin	homo	0.232 ± 0.005	0 ± 1
	Poisson	0.732 ± 0.006	0 ± 2
	normal	0.887 ± 0.006	1.5 ± 1.5
	random	1.051 ± 0.003	1.6 ± 0.7
	bimodal	1.322 ± 0.002	0 ± 0.5
Clustered Round Robin	homo	0.0154 ± 0.0004	2.9 ± 0.1
	Poisson	0.170 ± 0.005	13.2 ± 1.3
	normal	0.199 ± 0.001	5.1 ± 0.3
	random	0.663 ± 0.003	9.2 ± 0.8
	bimodal	0.686 ± 0.004	9.7 ± 0.9
Minimal Adaptive	homo	0.0199 ± 0.0008	2.7 ± 0.2
	Poisson	0.299 ± 0.005	10.8 ± 1.2
	normal	0.935 ± 0.008	9.1 ± 1.9
	random	1.344 ± 0.006	0.9 ± 1.4
	bimodal	1.81 ± 0.03	0 ± 7
First-come First-serve	homo	0.185 ± 0.002	0.8 ± 0.5
	Poisson	0.299 ± 0.003	5.1 ± 0.7
	normal	0.413 ± 0.007	1.7 ± 1.7
	random	0.754 ± 0.002	3.3 ± 0.6
	bimodal	0.763 ± 0.002	3.3 ± 0.6
Continual Adaptive	homo	0.0242 ± 0.0007	2.1 ± 0.2
	Poisson	0.0238 ± 0.0008	2.4 ± 0.2
	normal	0.151 ± 0.001	5.5 ± 0.3
	random	0.834 ± 0.003	11.8 ± 0.9
	bimodal	0.962 ± 0.006	12.4 ± 1.6

Table 5: Execution time per job (seconds) and job assignment latency overhead (seconds) as measured from experiments varying the number of jobs for each algorithm and job execution time distribution. The effective execution time is calculated by dividing the total execution time of the request by the number of processors used to execute the request. Job assignment latency was calculated by using slope- and intercept- figures from graphs of observed data. In some cases the data does not lend itself to the use of a Least-Squared Linear Fit of a Straight Line to produce job assignment latencies. This is found when the observed latencies are very small but have a large error, as in Round Robin homo, Round Robin Poisson and Minimal Adaptive bimodal.

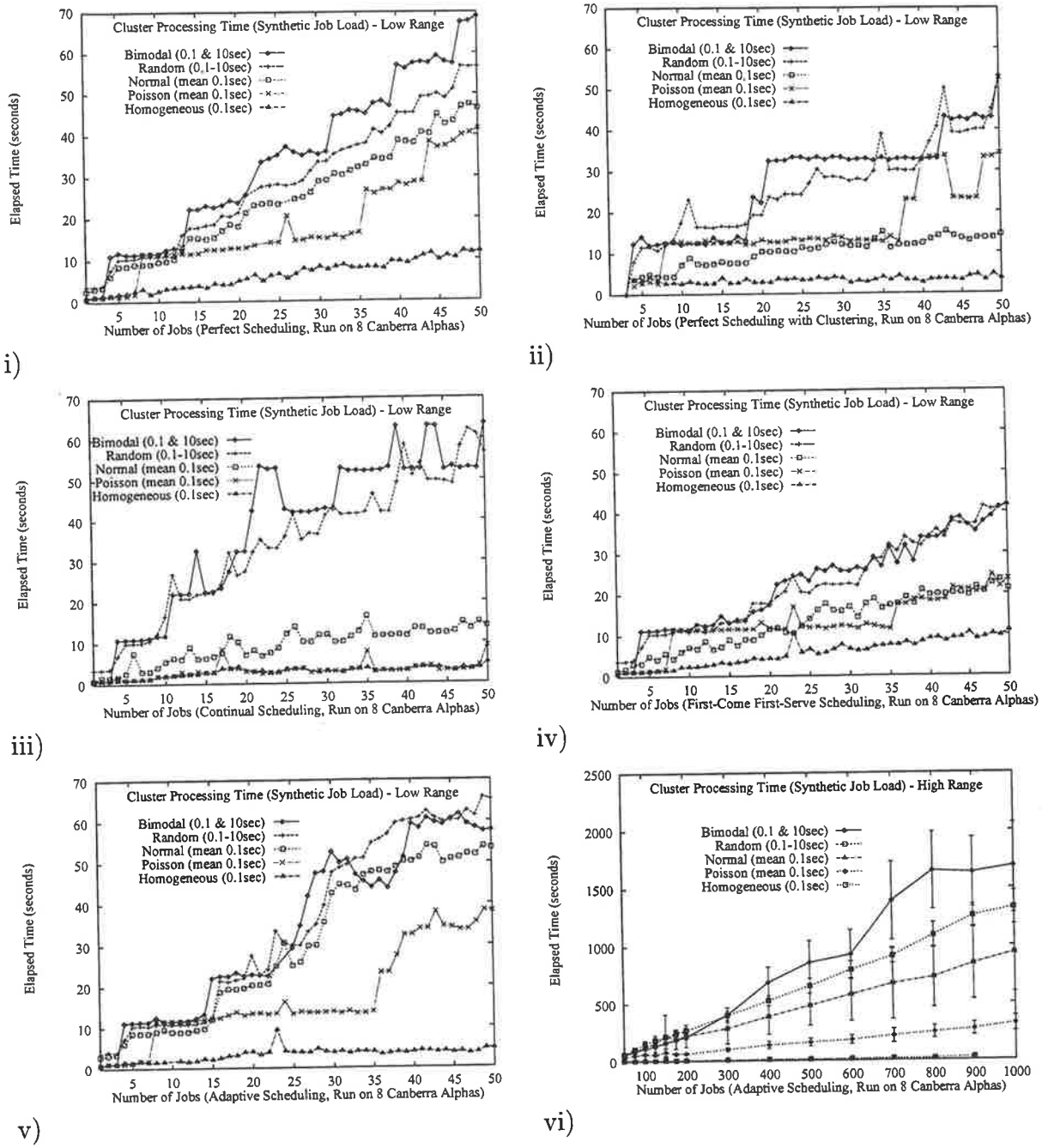


Figure 20: i)-v) Total execution time for 1-50 processes across 8 machines, vi) Execution time for 50-1000 processes across 8 machines.

round-robin scheduling, figure 20 ii), are flatter, reflecting a single job assignment latency. The homogeneous and normal distributions perform the best, due to their symmetric clustering of values around the mean. The Poisson distribution, although tightly clustered around the mean, lacks the symmetry of the normal distribution, which is reflected by decreased relative performance. The algorithm performs poorly with random and bimodal job execution distributions due to their high variability. That the data series representing the bimodal distribution seems to 'step' is purely random. The elapsed time reported in the graph shows the *total* time taken by the simulation to run the number of jobs in parallel across the cluster. Timing was started when the jobs were assigned and timing was finished when all machines in the cluster had finished their assigned jobs.

The continual adaptive algorithm's performance, shown in figure 20 iii), is best with those distributions that are easily predictable — homogeneous, Poisson and normal, even though the algorithm incurs a job assignment latency for each processor after synchronisation. This is due to its ability to predict, with a greater degree of accuracy, the time that a certain number of jobs will take to run before synchronisation is necessary. The main source of performance degradation in this algorithm is the wasted time when processors are idle waiting for another to finish before synchronising. Bimodal and random, because of their high variability, are hardest to predict successfully.

The slopes representing job execution time distributions in the test of first-come first-serve algorithm, shown in figure 20 iv), are more tightly clustered than those of round-robin, clustered round-robin and continual adaptive. This is due to the opportunistic nature of the scheduling algorithm, where those distributions exhibiting a mixture of long and short jobs average out over many repetitions. Making no predictions on the execution time of future jobs, this algorithm exhibits the best performance out of all algorithms studied for uniform random and bimodal distributions of job execution time. Since there are no predictions, there are no penalties for incorrect predictions, unlike in the case of the continual adaptive algorithm.

The minimal adaptive algorithm, shown in figure 20 v), represents the minimal amount of effort that can be expended in order to make an estimate of the relative performance of remote processors, based on actual performance of a single job. Thus, in the absence of any load information, it provides a rough basis for processor comparison. This algorithm also takes far less time to make predictions than the

continual adaptive algorithm. The performance of the algorithm is similar to that of the continual adaptive algorithm with the exception that the predictions of Poisson and normal distributions, based on a single point, are far more inaccurate, increasing the total execution time. Thus the more predictable the execution time of a job is, the greater performance this algorithm will achieve. Exactly why the data series representing the Poisson distribution jumps after 35 jobs is not clear. This phenomenon, while not critical to the argument presented in this chapter, is deserving of further investigation.

In the limit of large job numbers, the performance of all the algorithms, across all the distributions scaled linearly with job numbers, assuming fixed number of available processing nodes. Most algorithms settled into a steady-state performance scaling regime, and algorithm performance can be ranked in the order of best to worst for the distributions: homogeneous, Poisson, normal, random and bimodal.

In the large job number regime, the minimal adaptive algorithm, shown in figure 20 vi), exhibits irregular behaviour for the bimodal job execution time distribution, even for relatively large numbers of jobs. This is due to the algorithm failing to predict the job execution times properly, which is its major deficiency, but an inherent problem with a bimodal job time distribution.

The difference between the performance of the minimal and continual adaptive algorithms is insignificant in the case of a homogeneous distribution of job execution times. However, as the predictability of job execution times becomes more difficult, the benefit of collecting more execution-time data becomes evident.

In summary, if the distribution of job execution times is known in advance, as in the homogeneous case, then the algorithm with the least amount of job assignment latency and prediction overhead, clustered round-robin, is most suitable. For those distributions exhibiting symmetry and a high degree of clustering around the mean, such as the Poisson and normal distributions, a continual adaptive algorithm is most suitable. The first-come first-serve algorithm performed the best with those distributions that were hardest to predict, uniform random and bimodal.

Figure 21 shows summaries of real data collected by the ANU Supercomputer Facility [13, 14]. The first three graphs in the figure show that observed data can be seen to fit the distributions that we have used in this study. Furthermore, it can also be seen that in some cases (figures 21 iv) - vi)), the distribution of real data is not always predictable. Figure 21 iv) seems to be a combination of a homogeneous job mix with a relatively insignificant random distribution of turnaround times. Figure 21 v) seems

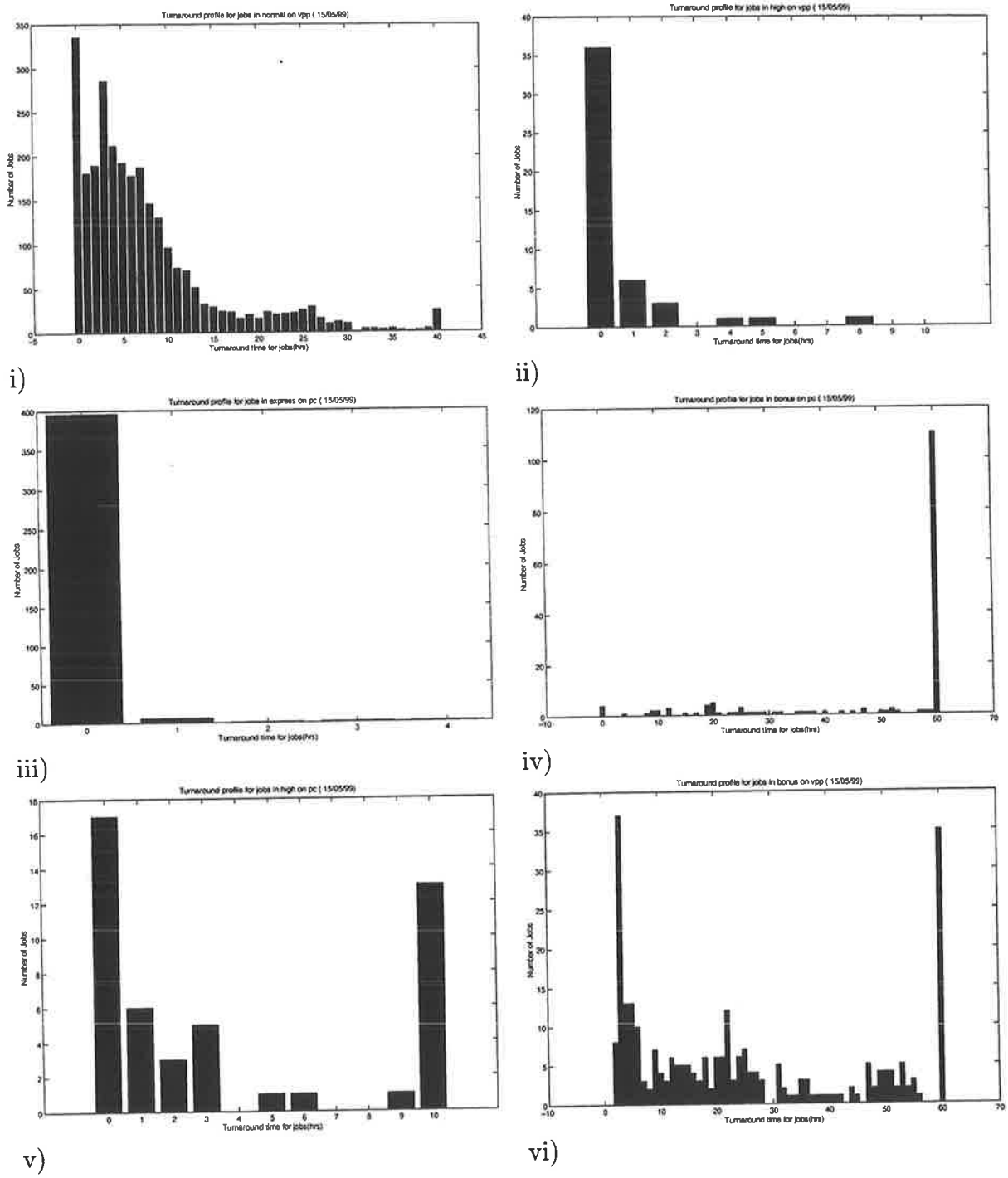


Figure 21: Job turnaround data collected from the ANU Supercomputer Facility. Users' jobs were submitted to one of a number of queues on a Fujitsu VPP300 and Silicon Graphics Power Challenge XL. Graphs i)-iii) show that the distributions used in our study are found in real data; graphs iv)-vi) show that real data is not always predictable.

to be a Poisson distribution with an equally significant homogeneous distribution. Of course, statistical analysis might reveal that the data actually conforms to a strong bimodal and weak normal distribution. Figure 21 vi) seems to be a bimodal distribution overlaid with a less significant Poisson distribution.

For those cases where the observed data neatly fits a particular distribution, the choice of scheduling algorithm is simplified. In the cases where the observed data is a combination of distributions, the decision of which algorithm to use can be quite complex. For example, if the data seems to be a combination of Poisson and bimodal distributions, then the two most appropriate algorithms that we have studied are the continual adaptive and first-come first-serve. What the scheduler needs to decide, in this case, is whether choosing the algorithm with the least amount of job assignment latency will be better than the ability to assign jobs to processors as soon as they have finished the current job. Certainly, as the number of jobs increases, the job assignment latency becomes more significant.

4.5 Discussion and Conclusion

The functionality of being able to distribute parameterised sections of a worker/slave program is not new, existing in selected applications, for example the `mpeg_encode` program [87] from Berkeley. This application has the facility to load balance across multiple UNIX hosts using one of two scheduling strategies: perfect (round-robin), which assumes each machine can process frames at the same rate, and assigns an equal fraction of frames to each processor; and a minimal adaptive algorithm, where a number of test frames are encoded to gauge the relative (static) performance of a remote processor and thereafter, a proportional number of frames is assigned to ensure that execution time of the frames is close to constant over all the remote processors.

Scheduling jobs on a general metacomputing system can be tackled by modeling the system as a hierarchy of sub-clusters. Each cluster of nodes, can be managed by a gateway node with the expected performance and ease of implementation as we have described. Clusters of these sub-clusters can potentially be managed using one or more of the existing scheduling software systems, where each managed node in the cluster is a gateway to a sub-cluster.

In general, we believe the terminology for static and dynamic scheduling is not entirely appropriate in the metacomputing context. For example, those algorithms

perform process placement at run-time but from a centralised scheduler with minimal knowledge of the remote processors do not fall neatly into either the categories of static or dynamic scheduling.

The Globus toolkit [69] and Legion [94] are two metacomputing projects which exhibit diametrically opposite approaches to resource management and process execution. Globus uses third-party schedulers, and users can submit arbitrary binaries for execution, while Legion is based on an object-oriented run-time environment and all user processes comprise of objects which extend a Legion base class. Thus, the Globus toolkit can use far less execution time information than is Legion. Using the hierarchical scheduling approach described in this chapter, we intend that our DISCWorld metacomputing system will be able to reap the benefits of both approaches. This is described in chapter 5.

The results have shown that for job execution times with a high degree of symmetry around a mean, a continual adaptive scheduling algorithm is most suitable, while for those distributions which are not easily predictable, a first-come first-serve algorithm performs most efficiently. We have shown that the job execution time distributions used in this study occur in real data, and that it is not uncommon for real data to be composed of a number of distributions.

We have shown that no one scheduling algorithm consistently produces the best performance across the job execution time distributions we have studied. Therefore it is important to know more about the actual distribution so that an appropriate algorithm can be chosen. Through the collection and exploitation of historical job execution times, we anticipate it may be possible for a *smart* scheduler to select and adapt to the most appropriate scheduling policy.

The main conclusion to arise from this study is that for the general case in which the job execution times are not known in advance, an adaptive algorithm, such as first-come first-serve or continual adaptive, is required. When the number of jobs to be scheduled is large, and the job assignment latency would be prohibitive to incur upon starting each job, a continual adaptive scheduling algorithm is the best choice. In order for the continual adaptive algorithm to be able to adapt responsively to changing conditions, more information than was used in this study needs to be collected.

Due to the differences in processing speed and machine characteristics (such as floating point arithmetic, processor time quanta, and available memory for multiple processes to run simultaneously) in a heterogeneous mixture of workstations, we

expect similar, but more exaggerated, results from the experiment. We expect that the adaptive algorithms, especially continual adaptive will perform the best due to its ability to favour the processors that have the best job throughput.

If the jobs which are submitted to the scheduler program as part of a larger computation, it may be possible to supply the scheduler with extra information to describe the jobs. This is the approach taken in our prototype DISCWorld metacomputing environment. A model for scheduling in coarse-grained metacomputers is presented in chapter 5; the implementation of the model is presented in chapter 6, in which the benefits of adaptive scheduling are shown.

The work presented in this chapter has three important uses: it provides a framework in which parameterised simulations can be run over a network of workstations; it provides a framework for the testing of scheduling algorithms on different types of job inputs; and it can be used as a service in the DISCWorld metacomputing environment to control a network of workstations on which it is not possible to execute a Java virtual machine.

Chapter 5

Modeling Scheduling and Placement

In this chapter we consider the problem of deriving placement information to schedule a program graph (as defined in section 3.3) consisting of services and the sharing of raw and result data. Unlike nearly all of the studies to be found in the literature (see chapter 3), the aim of this work is not simply to produce an algorithm to provide the best task-to-processor mapping in the situation where *all* the processors can execute *any* of the tasks in the user's job. This work differs from that discussed in section 3.3 in that not all tasks may be executed by any processor. In addition to the case where the task (or service to be used) may not be available on all processors (or nodes) in the system, we explicitly consider the case where the data needed for the services is not available on every node (and also may not be moved to where the services currently reside).

This chapter describes the model of the placement mechanism developed for use in our prototype metacomputing framework. This model allows a static mapping to be made between the services that are required in a processing request, and the machines that host them. Once the mapping is made and the system starts execution, the mapping can be dynamically modified to take into consideration additional system state information. The approach is a hybrid between static and dynamic scheduling, which is described more fully in section 3.1.3. The implementation of the metacomputing framework is described in chapter 6.

5.1 Terminology

For the sake of self-containment and completeness, we define our usage of some basic terminology. The *metacomputing daemon* is the software that controls the local resource in the context of the metacomputing environment, granting and denying users access, executing and monitoring jobs that are run on behalf of the users and the system as a whole. We use the terms *node* and *dumb node* to refer to a machine in the heterogeneous collection of computers that makes up a metacomputing system. We draw an important distinction between the two terms, however, insomuch as *node* is used to represent a machine upon which a copy of a *daemon* is run, and *dumb node* for the machine on which the daemon cannot run, for a given reason (such as an inability to run a Java Virtual Machine).

We use the term *job* or *task* to represent an application in execution, as known by the operating system. A job, in turn, comprises of a set of threads that execute together to form an application. As in [62], the exception to the term job is when a set of *processes*, which are autonomous as far as the operating system is concerned, are actually part of a single application. In this case, the operating system regards each process as a separate job. An example of this case is the concurrent execution of code on different machines using message passing constructs for communications. Each process is logically separate as far as the machines' operating systems are concerned, but together they comprise a single job. When we describe programs, the terms "job", "process" and "task" are used interchangeably.

5.2 Features and Constraints of Model

When modeling scheduling, it is beneficial to bear in mind a list of desirable attributes of the final model. In this section, the desired attributes and features of our scheduling model are discussed. These attributes of our scheduling model are: distributed scheduling; ability to schedule task graphs; ability to characterise processing nodes; restricted placement of services; heuristic optimal schedules; non-preemptive scheduling; and clustering of services onto nodes.

Some of the features of this model are: it is designed to perform in the absence of complete *up-to-date* system state information; and it is designed to schedule high-level programs which communicate by the sharing of *complete* results in producer – multiple-consumer relationships. The attributes and characteristics are explained more fully in the following subsections.

5.2.1 Distributed Scheduling with Partial System State

The distributed environment we consider is that of a loose confederation of distributed computers, separated by significant (in terms of network connectivity) distances. Furthermore, we consider the situation in which a user submits a processing request from a client application to its *local* DISCWorld node. In most cases, we expect the local DISCWorld node to be listening on a different port of the same physical machine, although it is not infeasible for the local DISCWorld node to be a different machine. Thus, the descriptive term *local* only refers to the proximity of the node to the machine running the client application, not necessarily the same physical machine.

The use of a centralised scheduler requires that all processing requests are sent to a *master node* for scheduling. In the case where the master node has a complete view of the global system state (see section 3.4), scheduling decisions can be made to optimise the execution time of the request or the effective utilizations of distributed resources. When using a centralised scheduler, each node can be instructed to send regular state updates to the scheduler. Because the system under consideration exhibits significant interconnection network distances, wide geographical separation of resources (see Appendix A for a discussion of this), and because the system is comprised of a loose confederation of machines, we are unable to mandate the use of a centralised scheduler.

Distributed scheduling, then, is another approach to scheduling that may be taken. In this approach, each node can make individual decisions on where the processing requests will be executed. It prevents the need for constant communication with a master node, and reduces the chance of a central point of failure for the system. Consequently with nodes making their own decisions of where to place services, the entropy of the distributed system may be increased. Of course, in the case where complete system state information is available, nodes are fully aware of the actual load on remote servers, and continual load balancing, with a high degree of accuracy, can be performed. In the environment under consideration, we are unable to guarantee that each node in the distributed system will have the same view of the global system state. Each node relies on its own view of the global system state – each server has its own view of which nodes are available, what services are hosted by each node, and what characteristics the other nodes can offer the local server. This incomplete view of the system is called *partial system state*, and is discussed in section 3.4.

The effects of partial system state are twofold: first, there may be machines participating in the distributed system which are unknown by other machines; and

secondly, the most up-to-date state information about a remote machine that *is* known, may not be available. The first case is shown in figure 22, where Server 1 knows about the existence of Server 5; Server 2 knows about Servers 1 and 5; Server 3 knows about Servers 4 and 5; Server 4 knows about Server 3; and, Server 5 knows about Servers 1 and 3. It can be seen that Server 2 does not know about Servers 3 and 4. The second case occurs because machines in the system are not static; the load on machines, and the data they possess are continually changing. Without complete system state information, the node that creates execution schedules must use the most up-to-date information it possesses. If this information is not completely accurate, then any schedule will not be optimal, however good the system state information is.

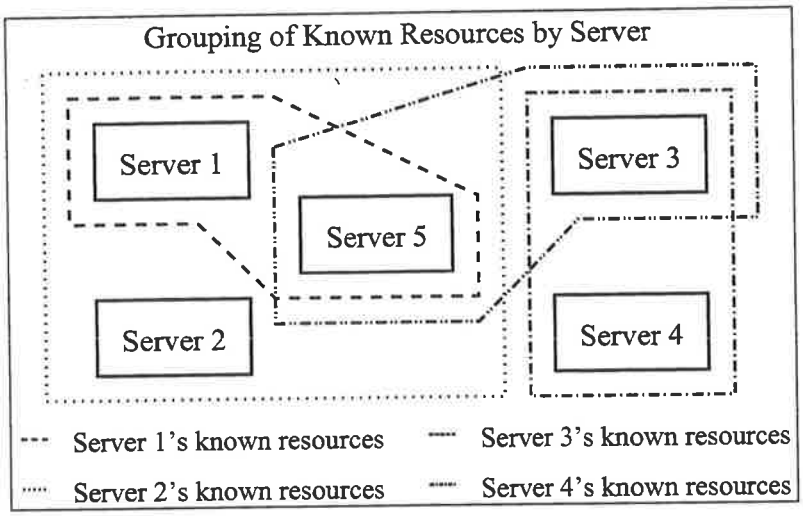


Figure 22: System state information may propagate throughout the system at different rates. When up-to-date global system state information is not available, partial system state information must be used as a basis for decision-making. For example, Server 1 might know only about services on Server 5; and Server 2 knows about both Server 1 and Server 5.

5.2.2 Model Inputs

Processing requests can be made by both user clients and servers. They are composed of graphs of high-level tasks, or *services*. Services are programs which can be run in the distributed system, and may be implemented as either platform-independent byte-code or platform-dependent applications. In addition, they may either be implemented as parallel or serial program code. The actual implementation of

services is not important to their execution by the DISCWorld daemon. Services are connected by data dependencies, which represent the use of data outputs of the producing service as inputs to the consuming service. Figure 23 shows the relationship between a processing request, services, data dependencies and producer-consumer relationships. We only consider high-level data dependencies; fine-grained parallelism between services is not supported, and data is not able to be used until the producing service has terminated.

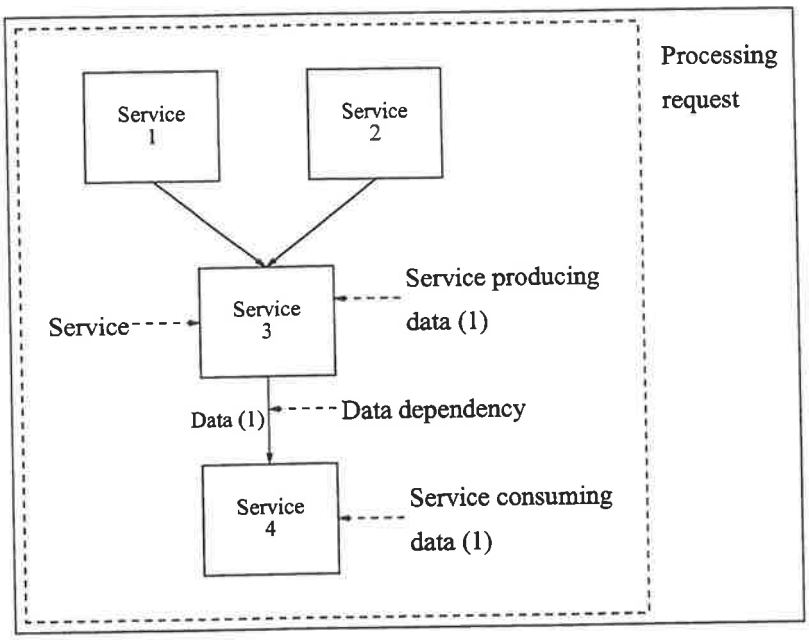


Figure 23: Relationship between DISCWorld processing request, services and data dependencies. A processing request consists of a number of services interconnected with data dependencies. The service creating the data is termed the *producer*, while the service that is to use the data is termed the *consumer*. Although it is not shown in this diagram, multiple services may consume the data produced by a producer.

Process networks [135, 136] are a common method of representing computations, where the inter-task dependencies are explicitly represented by edges between processing nodes. Process networks can specify parallelism between tasks, and can either be cyclic, which allows for the possibility of infinite iterations or acyclic, which do not. The subset of process networks, featuring directed edges and acyclic references are called directed acyclic graphs [8] (DAGs). DAGs are possibly the most common mechanism by which computations are represented in systems that reason about jobs

with inter-dependencies. They allow the flow of data and any precedence relations to be explicitly stated.

The name of a given service remains constant across the DISCWorld. Computations consist of a number of services that are chained together to create a potentially complicated sequence of operations. The presence of a loop in a graph implies the modification of data. DAGs are an ideal notation for describing the flow of control within a DISCWorld processing request because we restrict data to be write-once (or single-assignment). After data is created, any change to the data results on a new *derived* data item being created. For display purposes, however, we allow the use of loops in-conjunction with variable aliases (this is described in section 5.3).

Unlike the assumption made in [205], we are unable to constrain the DAGs to have a single root node. It is quite possible that there will be many extant data items that are to be used as input to services, which will need to be taken into account when devising an appropriate schedule.

We define a DAG as a tuple $G = (V, E)$ where $V = \{n_j, j = 1 : v\}$ is the set of task nodes (vertices, or services) and $v = |V|$ is the number of nodes, and E is the set of communication edges, implying a data dependency between the source and target nodes. $e = |E|$ is the number of edges. Thus, if each of the services is placed onto a different node, there may be a maximum of e data transfers in order to execute the schedule.

In the case under consideration, the presence of two or more outputs coming from a node does not represent a conditional branch, as in [55], but indicates the sharing of the results of the node. In our system, services - represented by nodes - can produce more than one output, each of which is a separate data item and any of which are able to be shared amongst more than one node or may not be used at all. Furthermore, if a service produces more than one output, more than one may be used by any number of services, requiring each to be individually transferred to the remote sever. For example, a service `Imagery:GMS_CloudCover`, which produces a cloud cover bitmap when supplied with a visual and infra-red GMS5 [133] satellite image of the same date-stamp, also creates a numeric figure corresponding to the percentage of the images. While the second output may not be of interest to the user submitting the processing request, it is still produced, and will, in all likelihood, be cached by the producing server, as it is non-trivial to reproduce.

5.2.3 Characteristics of System Components

In order to make sensible decisions on where to place services in order to achieve near-optimal execution time, and resource usage, it is vital that the components of the system are adequately characterised. For the purposes of this study, the relevant components of the system are: Data; Services; Nodes; and Interconnection Networks.

The fundamental assumptions of this scheduling model are: the existence of a **platform-independent global naming strategy**; and, a **single assignment restriction for data**. We assume that a mechanism for the platform-independent, canonical naming of data and services exists, and is adhered to by all participating users, nodes and services. In this context, the term *unique* refers to different data, not copies of the same data. When data is ingested into the system, or created by a service, it is named according to the global naming strategy. By restricting data to be single assignment, we prevent data from being changed. This restriction allows the simplifying assumption that if two objects have the same global name, they are the same. The execution of a service results in output data, which is named in accordance with the global naming strategy. We assume that the naming strategy is hierarchical, and that output data is named according to the service and input data used to create it. The global naming strategy used in the prototype DISCWorld implementation is discussed in section 6.2.

The remainder of this section discusses the minimum amount of information that must be held on each of the components in order to properly schedule processing requests. We require that each of the services be characterised, and need to bear in mind the information we will be given about the services may not be static or completely up-to-date. In addition we need the ability to collect the information about the services, data and nodes, and to incorporate that into the scheduling decisions that are to be made by the system.

Data Representation

In our scheduling model, we recognise two different types of data: *raw* data, which may be injected into the system by a physical device or a user; and *derived* data, which is the output of a service when executed with either raw or derived data. For the purposes of scheduling and placement, both are considered equivalent. We broadly characterise the data as being large, and expensive to produce; we also consider the

case where data belongs to an organisation and it available for use at a price.[1]

Data is stored on a node, which may be where the data was created, where it is about to be used, or a data repository. There may be multiple copies of the same data at different nodes within the system. Due to the assumed global naming strategy, each copy of the data will have the same name; hence it can be used interchangeably. We recognise two characteristics of data that are important for our scheduling model: size of data and mobility.

size of data The size of the serialized data object. The size of the data item is used when considering the option of moving the data to another node in the system.

mobility Ability for this data to be moved between nodes in the system. Where the data is sensitive or proprietary, data owners may not be willing to let the data be copied between nodes. If the data object is derived data from unmovable raw or derived data, the node owner may not mind it being copied.

Although the size of a data item is fixed and constant independent of where it is stored, the mobility of the same item may vary between nodes. For example, a node's administrator may stipulate that the data they own may be copied or moved at a cost premium. However, all those nodes that take a copy of the data are not able to pass the copy on but may still use it as inputs to services. These characteristics of the DISCWorld are termed *policies*. Policies are discussed in section 6.5.3. When a data item is to be used as an input to a service, the size of the data and its mobility will determine whether it is more feasible (in terms of time or economic units) to copy the data item to where the service presently exists, or to copy the service to where the data exists.

Service Representation

Services are high-level programs, which may be implemented as parallel or serial code, represented by platform-independent byte code or platform-dependent object code. They may be small applications which perform a general task, such as the cropping of images [129] or more specialised programs such as producing cloud cover classifications from satellite imagery [106]. They may even be larger legacy applications which have been optimised to run on specialised equipment [18,105]. Our scheduling model treats

[1] While there have been studies that consider economic issues [10,213,214], no discussion is made in this thesis.

services as *black boxes*, where their actual implementation is not important for their high-level characterisation.

As mentioned above, producer-consumer relationships between services in a processing request are at a high level. Fine-grained message passing is not supported; only complete outputs of a service are able to be shared. Due to the global naming strategy, equivalent services with the same names can be used interchangeably to perform a known function. The characteristics that we consider important for services are: size of byte code; service run time; service run time variance; and mobility.

size of byte code The size of the serialized byte code. This size also includes any necessary non-core or non-Java foundation classes, packaged up into a Java Archive file.

mean run time The mean run time of this service on the current node.

run time variance The variance of run time of this service on the current node.

mobility Ability for this service to be moved between nodes in the DISCWorld. Services that use node-specific resources, such as those that control certain storage devices, may not be mobile.

In contrast to the characteristics of a data object, there are no guarantees that characteristics of a service will be the same. The run-time information pertains to the instance of the service at a given node only. When a service is considered for transfer to a new node, we assume, for simplicity, that the estimated run-time of the service at the *new* node will be approximately the same as on the node from which it came. When the service is actually run on a new node, any run-time information from the previous node is discarded.

In order to schedule processing requests in a sensible way, it must be possible to profile the services that comprise the request. Profiling is the measuring of the performance and resource requirements of a service. There are a number of ways in which it is possible to profile services.

One method is by the collection of historical run-time data of a service. Whenever a service is run on the same machine, the execution time is recorded and the resources that it consumes are measured. Over time, the measurements may settle into a steady state, from which predictions can be made. If the service's execution times do not settle into a steady state, then the execution time may be best expressed as either a mathematical distribution or a mean and variance tuple (see section 4.2). If the

machines that a service may be run on are homogeneous, then the data collected from each of the machines individually may be collated to provide more data over which to make predictions (because the data is more statistically significant); care must be taken that data from heterogeneous machines, on which resource requirements may vary, are not combined.

Another method of predicting the run-time of a service is by the inspection of the source code. Such a source code complexity model is a relative measure, giving rise to an estimate of run time. Unfortunately, this is not a useful measure in the event that the service that is being executed is a wrapper for pre-compiled object code, as is usually the case with commercial programs. Because we treat services as 'black boxes', we do not use this method.

A popular method of predicting the run-time of services, used by most queueing systems is requesting that the user specify, as accurately as possible, an upper bound on the run-time of their service, or program. This places the onus of prediction onto the user. A range of incentives are used to encourage the user to accurately predict the run-time, such as allowing shorter jobs to be placed on faster processing queues, or automatically killing jobs that overrun their estimate by a certain amount.

Simulation is another method by which a user may estimate the run time and resource requirements of a program, and pass information onto the queueing system. It may be possible to run the program with a synthetic distribution of input data, generating average run times and resource usages of the synthetic jobs, before executing the real program on a machine that may be expensive, in terms of available CPU time or money.

One of the issues that must be addressed when attempting to profile a service is whether it is possible to parameterise the profile. For example if a service reads a constant amount of data from a given file, it is easy to predict future run time and resource requirements, or if it reads a variable amount of data, it may be possible to parameterise the run time with the amount of data that is to be read. If the data that must be read is dependent on some other condition, for example, it reads from a file until it has read a certain number of sparsely-placed tokens, then it may not be possible to easily parameterise.

One of the main problems with the collection of performance and resource utilizations data from programs is the impact that the collection may have on performance. As with deriving schedules, if it takes longer to collect the performance data and form the profile than it would have simply running the program on any

available machine, or if it is only intended that a program be run once only, then it is obviously not worth the effort of profiling.

In the system under consideration, the same processing requests will be repeated infrequently, while the services that comprise the requests are sufficiently general that they can be used quite often, in possibly different contexts. Thus, profiling data, while useful for services, is not anticipated to be very much use for complete processing requests. This approach is in direct opposition to other approaches, such as found in most batch queueing systems and wide-area managers, which lack the ability to inspect the composition of user-submitted jobs.

Node Representation

In our scheduling model, nodes are machines that run a metacomputing daemon. The metacomputing daemon allows the machine to perform scheduling operations and exchange data and services with the wider metacomputing environment. While running the daemon allows services to be run, and data to be created on behalf of the distributed system, the owner does not yield control of the resource. They still retain absolute authority over their resource.

This model assumes that all nodes participating in the distributed environment are trusted. Thus, a processing request arriving from a remote node will be honoured, providing the node has been configured to do so. We identify four characteristics that facilitate scheduling across the distributed system: the current waiting time to start the execution of a new service; the capacity to accept new service requests; the ability to accept new service byte-code; and the access costs for using a node.

current waiting time to start new services Instead of using a machine-specific measure of the load on the system, which will be dependent on the vagaries of the machine (such as multiple processor systems, and those machines that are continually fully loaded by batch queueing systems), each machine provides an estimate of the waiting time that a newly-submitted service *may* have to wait before beginning execution, providing all necessary input data is present on the daemon.

capacity to accept new service requests Independent of whether the node is currently loaded or unloaded, this attribute represents the willingness of the node to execute processing requests on behalf of users or remote servers.

ability to accept new service byte code Independent of the current load on the node, this attribute represents a node's willingness to accept uploaded byte code to perform a new service.

access cost The cost which is charged to access a node's services or data.

The characteristics representing a node's capacity to accept new service requests and to accept new service byte code are similar to the mobility of a service or data. If the node creating a schedule has a choice between placing a service on two or more remote nodes, then the node with the greatest capacity to accept new requests should be chosen. Similarly, if a copy of the service doesn't exist on any of the remote nodes, then the node with the greatest capacity for accepting new service byte codes should be chosen. A node's access cost is only incurred when it is the source of a data or service byte code transfer.

A node's current waiting time to start new services is an estimate based on the historical behaviour of the daemon (which actually runs the services). While the estimate may be not completely up-to-date at the present point in time, it serves to provide remote nodes with an idea of the response time of the remote node. Even though the above characteristics are primarily used to describe a *node* within the distributed system, a subset of the characteristics may be used to describe a *dumb node* in the system. We use the current waiting time to start a new service as the minimum characteristic of a node.

Interconnection Network Representation

We assume that the collection of processors that comprise the distributed system is a point-to-point fully connected set, in that while there may be no *direct* connection between a two machines, there will be a route between them (perhaps via the Internet or using a specialised broadband network [140]). Traditionally, there are two characteristics that define an interconnection network: bandwidth and latency.

Advertised bandwidth and latency characteristics are stored between pairs of nodes. Because the actual availability of a link, and the achievable bandwidth between two points is very variable [132], we only use bulk characteristics.

5.2.4 Restricted Placement

Our model differs from those previously discussed in section 3.1, in that some of the services and data may be unable to be placed on arbitrary processors. The location of

some services are fixed, most often due to their method of implementation: they may use a large amount of scratch space; they may be implemented in a fashion that uses a parallel node; or, they may require specific hardware that only exists on a subset of the nodes in the distributed system.

The scheduling model needs to take placement restrictions into account when creating schedules that involve heterogeneous machines in a distributed environment. The characteristics of the system components, listed in the previous subsection must be used to ensure that any placement of services and data is valid.

5.2.5 Heuristic Optimal Schedules

Our model seeks to minimise the communication volume, while at the same time minimising the parallel cost time across a **bounded number** of nodes. The parallel cost time is the additional cost (measured in increased processing time on a per-process basis) incurred when unrelated services that are able to be executed in parallel, are scheduled to run on the same node at the same time. Parallel cost time is not incurred when related services are co-located on a node, because the explicit temporal ordering imposed by producer-consumer relationships. While it is well known that scheduling tasks across an unbounded number of nodes in the general case is NP-complete, we restrict the number of nodes that are considered for placement, of either data or services.

We restrict the nodes that are considered for placement by only choosing those that either: already host a copy of the service that is to be used; already host the data that is to be used; or will receive a copy of the data or service in a previous processing step. Thus, we choose only those nodes which have at least partial satisfaction of the requirements for a service to be executed. The use of the system characteristics compliment the selection of viable nodes by further restricting the available set.

Of course, the use of decentralised scheduling and possible partial system state information may mean that a node is not aware of *all* remote nodes that host a service or have the required data. This is addressed in chapter 6, and in particular, section 6.1, where the mechanism for schedule execution is discussed.

In minimising the communication volume and parallel cost time, we require that the execution schedule thus created is *optimal* for the nodes that have been considered for placement. As such, we do not require that the created schedule be optimal for the *global system*, but instead that it is optimal for a restricted sub-set of considered nodes.

5.2.6 Duplication of Services and Data

One of the features of our scheduling model is that of the possibility of duplication of services and data. Such duplication may occur on two levels: the services and data that are to be used may be duplicated across nodes in the distributed system; and, multiple instances of the same combination of service(s) may be executed in order to reduce the overall processing request execution time.

At present, we only support the first type of duplication: that of multiple sources for services and data. Because the creation of execution schedules can be performed by any node within the system, and because nodes are under no obligation to cooperate, the same data may eventually be created on multiple nodes. As discussed in section 6.1, if a service requires data that is available from multiple sources, a mechanism exists to choose the optimal source.

While we fully intend on doing so, at present we do not consider the multiple execution of the same service(s) on different nodes in order to reduce the overall execution time, as in [205]. Such a modification would not be complicated, and its impact on the system would be of interest. Presently, the unit of cost to use the system is time. When users are required to "pay" for the resources they consume, the use of multiple resource that may provide a faster request resolution will introduce trade-offs between the time they wish to spend waiting for a request to be resolved, and the amount of resources they wish to be used on the request.

5.2.7 Clustering of Services

The method by which schedules are created to minimise both communication volume and parallel cost time is through the clustering of services, as in [40,86,205]. Services that share a large amount of data are placed onto the same node, or nodes with high-speed interconnection networks. Clustering is discussed in section 3.3.

Clustering algorithms group services together onto a single node to reduce communications delays. When the schedules are executed, clustering approaches such as [86] serialized the execution of services which may be run in parallel. Thus, they assume that the processing hardware is only able to execute a single service at a time. While this approach is adequate for systems in which there exists a centralised scheduling mechanism or complete system state information, the system under consideration in this study has neither.

Their approach was thus one of a brute force attack, considering the placement of each service on every node. Through the using restricted placement of services, we constrain the complexity of the search that must be made to achieve an optimal solution to the constrained case that we consider.

5.2.8 Non-Preemptive Scheduling

In our scheduling model, we assume that tasks are scheduled in a non-preemptive manner. In this case, when a service begins executing with all its required inputs, it will continue executing until it terminates, without interruption. While it is quite probable that the daemon, under which the services will be executed, will execute tasks in a co-scheduled fashion [175], this level of detail is too fine for the high-level schedules considered here.

If preemptive scheduling is allowed, then other threads of control will be executed together with the currently-executing service. The nett effect of the multi-threading will be an increase in expected execution time. In order to keep the scheduling model as simple as possible, yet applicable to the real system, we only assume that a single service is executing at a time. In some ways, the characteristics that we maintain on a service reflect the case in which the service is executed together with others. Multiple services being executed simultaneously will cause an increase in both the mean run time, and variance in the run-time for the services running on that node.

5.3 A Distributed Job Placement Language

One of the fundamental mechanisms that separates metacomputing environments from most batch queueing systems or simple client-server systems is the ability to specify a number of operations that are to be performed on some data, and have the operations individually scheduled. This can be done without the intervention of the user at every step of the processing, whether that intervention is the initialisation of the next processing step, or the submission of raw or derived data to a new program. We term the specification of operations to be performed a *processing request*. The mechanism used to express processing requests in the DISCWorld prototype is the Distributed Job Placement Language (DJPL) [107].

This section describes the DJPL, and how it is used to specify services on behalf of the client, whether the client is a human user or another DISCWorld daemon. The DJPL grammar is presented and an example is given of its use.

In the DISCWorld metacomputing environment, processing requests are expressed in the form of services on data, arranged as a directed acyclic graph. When a processing request is submitted by a user client or remote daemon, the services that comprise the request have not been assigned, or *placed* onto nodes. We term this an *un-annotated* DJPL script. When the services have all been assigned to nodes, we term the resulting DJPL script, *annotated.* An annotated DJPL script may be annotated using the placement model described in section 5.4 and executed by DISCWorld daemons (as discussed in chapter 6).

The DJPL is designed to encapsulate all the information that is necessary for a client's request to be executed by a daemon within the DISCWorld environment. The client can be either a human user or another DISCWorld daemon that is making an execution request. The information that is needed by the DISCWorld daemon in order to execute the request is:

user data data to identify the user in the DISCWorld context. Includes information as to the user's identity, which group they belong to, and any access restrictions on the data that may be produced as a product of the request.

job data information to identify the job, such as the originating server or client's ID, the priority of the job, the preferred and absolute maximum time limits or costs.

aliases in the case where the global names given to data or services are too long to be easily read by humans, aliases are provided to 'factor out' common elements in global names. This has the effect of making the scripts easier to read by humans, although direct substitution is used upon the script arriving at a DISCWorld scheduler/parser. This mechanism is very similar to that of variable substitution in Unix command shells.

instruction stream the command structure of the request, and the services and data that are needed. In addition, the names of DISCWorld nodes that have been chosen to implement a partial or complete schedule are inserted into the instruction stream.

DJPL Grammar

To simplify the task of parsing the DJPL script, it is expressed in XML [238]. Although transmitting a pre-parsed, object-version of the processing request would

be more efficient, this decision allows us to use an architecture-independent representation for requests, and makes the DJPL easy to read by humans. Free XML parsers for Java are also readily available [124, 145]. Figure 24 shows the XML document template definition (DTD), which is used to specify the manner in which the DJPL script is written.

For the purposes of this thesis, the most important section of the DJPL is the **instruction** section. In this section, the services that have been used to decompose the processing request are itemised, and the data they consume and produce are named. If any aliases are defined in the **alias** section, they are directly substituted into the instruction stream at execution time. Loops, iterating over a pre-defined range of values, are allowed in the instruction stream. They provide a short-hand way of repeating the same instructions across a range of values. This is implemented by a temporary alias, which is substituted into the instruction stream. We impose the restriction that there must be no data dependencies between iterations of the loop.

An Example DJPL Script

Figure 25 shows a DJPL script that describes a processing request. A loop is used to iterate over the day range `Integer:01` to `Integer:30`. An `Imagery:Crop` service is used to crop the visual and infra-red channels of the GMS5 satellite images corresponding to those days. For each pair of cropped output images, the service `Imagery:GMS_Classify` is executed. On the output of this service, the final service `Imagery:Georectify` is executed. For further explanation of this processing request, see [130].

Expressing a query in using DJPL is similar to, but significantly different from the *classad* [192] structure as used in Condor and the resource specification language (RSL) [48] used in Globus. Whereas the classad structure and RSL define a set of constraints that the target must satisfy to execute a job, the DJPL is designed as a general structure, from which an execution schedule may be generated. The Helios operating system [185] featured a similar language for shell scripting, which allowed multiple programs to accept inputs from a single producer, and the Computing Center Software project used a resource definition language [22] to represent Transputer configurations for parallel programming.

Defining the language in XML allows it to be extensible. If a daemon receives an XML element that its parser does not understand, it is able to ignore it. We are

```
<?xml encoding="US-ASCII"?>

<!ELEMENT DJPL (USER,JOB,ALIASES?,INSTRUCTIONS)>
  <!ELEMENT USER (ID,GROUP?,PERMISSION?)>
    <!ELEMENT ID (#PCDATA)>
    <!ELEMENT GROUP (#PCDATA)>
    <!ELEMENT PERMISSION (#PCDATA)>
  <!ELEMENT JOB (JOBID,PRIORITY,COST,TIME,SERVER)>
    <!ELEMENT JOBID (#PCDATA)>
    <!ELEMENT PRIORITY (#PCDATA)>
    <!ELEMENT COST (SOFT?,MAX,ESTIMATE?)>
    <!ELEMENT SOFT (#PCDATA)>
    <!ELEMENT MAX (#PCDATA)>
    <!ELEMENT ESTIMATE (#PCDATA)>
    <!ELEMENT TIME (#PCDATA)>
    <!ELEMENT SERVER (NAME,PORT)>
      <!ELEMENT NAME (#PCDATA)>
      <!ELEMENT PORT (#PCDATA)>
  <!ELEMENT ALIASES (ALIAS)+>
    <!ELEMENT ALIAS (NAME,VALUE)>
      <!ELEMENT VALUE (#PCDATA)>
  <!ELEMENT INSTRUCTIONS (INSTRUCTION|FOREACH)+>
    <!ELEMENT FOREACH (VARIABLE,RANGE,BODY)>
      <!ELEMENT VARIABLE (#PCDATA)>
      <!ELEMENT RANGE (#PCDATA)>
      <!ELEMENT BODY (FOREACH?,INSTRUCTION+)>
    <!ELEMENT INSTRUCTION (NODE?,SERVICE,PARAMETER+)>
      <!ELEMENT NODE (#PCDATA)>
      <!ELEMENT SERVICE (#PCDATA)>
      <!ELEMENT PARAMETER (NAME,VALUE)>
      <!ATTLIST PARAMETER TYPE CDATA #REQUIRED>
```

Figure 24: DJPL XML document template definition (DTD)

```
<?xml version="1.0"?>
<!DOCTYPE DJPL SYSTEM "djpl.dtd">
<DJPL> <USER> <ID>heath</ID> <GROUP>dgis</GROUP>
            <PERMISSION>unrestricted</PERMISSION> </USER>
       <JOB> <ID>00005</ID> <PRIORITY>5</PRIORITY> <COST> <SOFT>100</SOFT>
             <MAX>120</MAX> <ESTIMATE>90</ESTIMATE> </COST> <TIME>360</TIME>
             <SERVER><NAME>opal.cs.adelaide.edu.au</NAME><PORT>6668</PORT>
             </SERVER> </JOB>
  <ALIASES>
    <ALIAS> <NAME>vis_image</NAME>
      <VALUE>Image:GMS5(Integer:0, Integer:00, $date, Integer: 09, Integer:98)
      </VALUE> </ALIAS>
    <ALIAS> <NAME>ir_image</NAME>
      <VALUE>Image:GMS5(Integer:1, Integer:00, $date, Integer: 09, Integer:98)
      </VALUE> </ALIAS>
    <ALIAS> <NAME>lower_bound</NAME> <VALUE>Integer:0</VALUE> </ALIAS>
    <ALIAS> <NAME>upper_bound</NAME> <VALUE>Integer:1024</VALUE> </ALIAS>
  </ALIASES>
  <INSTRUCTIONS>
    <FOREACH>
      <VARIABLE>date</VARIABLE>
      <RANGE>Integer:01 Integer:02 Integer:03 Integer:04 Integer:05 Integer:06
             Integer:07 Integer:08 Integer:09 Integer:10 Integer:11 Integer:12
             Integer:13 Integer:14 Integer:15 Integer:16 Integer:17 Integer:18
             Integer:19 Integer:20 Integer:21 Integer:22 Integer:23 Integer:24
             Integer:25 Integer:26 Integer:27 Integer:28 Integer:29 Integer:30
      </RANGE>
      <BODY>
        <INSTRUCTION> <SERVICE>Imagery:Crop</SERVICE>
          <PARAMETER TYPE="INPUT"> <NAME>input_image</NAME> <VALUE>$vis_image</VALUE> </PARAMETER>
          <PARAMETER TYPE="INPUT"> <NAME>upperleft_col</NAME> <VALUE>$lower_bound</VALUE> </PARAMETER>
          <PARAMETER TYPE="INPUT"> <NAME>upperleft_row</NAME> <VALUE>$lower_bound</VALUE> </PARAMETER>
          <PARAMETER TYPE="INPUT"> <NAME>lowerright_col</NAME> <VALUE>$upper_bound</VALUE> </PARAMETER>
          <PARAMETER TYPE="INPUT"> <NAME>lowerright_row</NAME> <VALUE>$upper_bound</VALUE> </PARAMETER>
          <PARAMETER TYPE="OUTPUT"> <NAME>output_image</NAME> <VALUE>Imagery:Crop:output_image($vis_image, $lower_bound,
          $lower_bound, $upper_bound, $upper_bound)</VALUE> </PARAMETER>
        </INSTRUCTION>
        <INSTRUCTION> <SERVICE>Imagery:Crop</SERVICE>
          <PARAMETER TYPE="INPUT"> <NAME>input_image</NAME> <VALUE>$ir_image</VALUE> </PARAMETER>
          <PARAMETER TYPE="INPUT"> <NAME>upperleft_col</NAME> <VALUE>$lower_bound</VALUE> </PARAMETER>
          <PARAMETER TYPE="INPUT"> <NAME>upperleft_row</NAME> <VALUE>$lower_bound</VALUE> </PARAMETER>
          <PARAMETER TYPE="INPUT"> <NAME>lowerright_col</NAME> <VALUE>$upper_bound</VALUE> </PARAMETER>
          <PARAMETER TYPE="INPUT"> <NAME>lowerright_row</NAME> <VALUE>$upper_bound</VALUE> </PARAMETER>
          <PARAMETER TYPE="OUTPUT"> <NAME>output_image</NAME> <VALUE>Imagery:Crop:output_image($ir_image, $lower_bound,
          $lower_bound, $upper_bound, $upper_bound)</VALUE> </PARAMETER>
        </INSTRUCTION>
        <INSTRUCTION> <SERVICE>Imagery:GMS_Classify</SERVICE>
          <PARAMETER TYPE="INPUT"> <NAME>input_vis</NAME> <VALUE>Imagery:Crop:output_image($vis_image, $lower_bound,
          $lower_bound, $upper_bound, $upper_bound)</VALUE> </PARAMETER>
          <PARAMETER TYPE="INPUT"> <NAME>input_ir</NAME> <VALUE>Imagery:Crop:output_image($ir_image, $lower_bound,
          $lower_bound, $upper_bound, $upper_bound)</VALUE> </PARAMETER>
          <PARAMETER TYPE="OUTPUT"> <NAME>output_image</NAME> <VALUE>Imagery:GMS_Classify:output_image(
          Imagery:Crop:output_image($vis_image, $lower_bound, $lower_bound, $upper_bound, $upper_bound),
          Imagery:Crop:output_image($ir_image, $lower_bound, $lower_bound, $upper_bound, $upper_bound)) </VALUE> </PARAMETER>
        </INSTRUCTION>
        <INSTRUCTION> <SERVICE>Imagery:GMS_GeoRectify</SERVICE>
          <PARAMETER TYPE="INPUT"> <NAME>input_vis</NAME> <VALUE>Imagery:Crop:output_image($vis_image, $lower_bound,
          $lower_bound, $upper_bound, $upper_bound)</VALUE> </PARAMETER>
          <PARAMETER TYPE="INPUT"> <NAME>input_ir</NAME> <VALUE>Imagery:Crop:output_image($ir_image, $lower_bound,
          $lower_bound, $upper_bound, $upper_bound)</VALUE> </PARAMETER>
          <PARAMETER TYPE="OUTPUT"> <NAME>output_image</NAME> <VALUE>Imagery:GMS_GeoRectify:output_image(
          Imagery:Crop:output_image($vis_image, $lower_bound, $lower_bound, $upper_bound, $upper_bound),
          Imagery:Crop:output_image($ir_image, $lower_bound, $lower_bound, $upper_bound, $upper_bound)) </VALUE> </PARAMETER>
        </INSTRUCTION>
      </BODY> </FOREACH> </INSTRUCTIONS>
</DJPL>
```

Figure 25: Example DJPL script before annotation. Services, and the parameters that are to be used are named to allow a data flow graph to be constructed. Outputs of each service are also named to allow re-use.

currently investigating methods of specifying the criticality of XML elements, so that if a parser does not understand a critical element, it will terminate parsing.

We are also investigating methods of encoding exception-handling instructions into the DJPL. If an exception occurs, for example, if a service is unable to be found on any of the known servers, then the user may want to be immediately notified, or may just want the maintainer of their local DISCWorld daemon notified.

We are currently experimenting with annotating the DJPL code in such a way as to instruct it not to further distribute the script if it is unable to provide the service. Alternatively, we could annotate it in such a way as to record all the daemons that have sent it on, thus preventing the case where daemons simply pass the request among themselves, never making forward progress.

The DJPL comprises an important part of the DISCWorld scheduling and placement model. While designed and implemented specifically for use with DISCWorld, the language itself is general enough to be used in other metacomputing or middleware environments.

5.4 Scheduling Model

In the literature, a task is defined as an indivisible unit of computation, and the tasks are convex, which means that once a task starts its execution, it can run to completion without interruption [201]. We modify the definition of a task slightly so that while it is still the unit of computation, it may itself represent a number of tasks that have been grouped together under a single name.

As discussed in section 3.1, there are two main models for allocating tasks to processors: static and dynamic. Neither approach is adequate in the case where limited system state, or at least the possibility that out-of-date global system state, is available to a scheduler.

When reference is made to two services, they may either be termed *related* or *unrelated*. The term related means that one of the services is a consumer of the output data produced by either the other service or a consumer of the service. If two services are related, there is an implicit ordering of execution between them; they are unable to be executed at the same time. The term unrelated means that there is no producer-consumer relationship between the two services, and hence, they may be executed at the same time.

The problem of disseminating global system state information may be approached by defining a strict hierarchy of nodes, or state brokers, from which the most recent global state information may be retrieved. Of course, the state broker for a cluster of nodes will have its own state broker that supplies it with global information on the remainder of the distributed system. There must exist a two-way information flow such that a nodes' broker can be updated with the nodes' information at the same time as the nodes can retrieve information on the remainder of the system.

We feel that this approach to disseminating system state information is inappropriate for our model. This is true for a number of reasons: there is too much system information that must be shared; nodes are truly dynamic in their availability; and, we are unable to define a strict hierarchical relationship that all resource owners are likely to agree upon. For these reasons, we have decided to adopt a system whereby advantage is taken of partial system information, and nodes are updated through the receipt of typical information and control messages.

In our model, the nodes' `accept new jobs` and `accept new service byte code` characteristics are represented by the range values $[0, 2]$. The value of 0 is used to indicate that the characteristic is not supported by the node; 2 is used to indicate full support of the characteristic, with no cost penalty; and, 1 is used to indicate that the characteristic is supported reluctantly, with a cost penalty for use. The cross-product of the possible values for a node's *accept new jobs* and *accept new service* characteristics is shown in table 6. A description and possible example of each combination is given. Where the willingness of a node is represented by a value, this does not suggest the use of probabilities in decision-making. These factors are used in the cost functions to weight the decisions in order to properly use an objective function to make an informed decision.

5.4.1 Cost-Minimisation Function

The state of the system is stored on a per-node basis. Each node in the distributed environment collects and uses information as to the state of the rest of the system. The collection of information as to the remainder of the system allows us to make restricted placement decisions of jobs, with more information and more nodes being able to be incorporated as the system becomes *aware* of other nodes. The collection of system state knowledge alleviates the need for each node to be aware of every other node, and also alleviates the need for designated *control nodes* or *brokers*, which other systems feature. This allows for a truly dynamic system to be constructed, where

Accept New Service Byte Code	Accept New Service Request		
	0	1	2
0	data repository only. No request execution	mainly data repository with reluctant service execution	data repository welcoming processing requests
1	reluctantly archive new service. No request execution	reluctantly accept and execute services	reluctantly accept new services. Gladly execute any services
2	advertised service archive. No request execution	new services welcome with reluctant execution	new services welcome. Gladly execute any services

Table 6: Description and examples of node *accept new service byte code* and *accept new service request* characteristics cross-product

membership in the confederation of cooperating nodes is based on the current state of each node. In other systems, if a node that was designated as a control node suffers a failure, the remainder of the system will be unable to access the characteristics of the nodes that the control node manages. In our truly decentralised system, information is propagated by a protocol, which is outwith the scope of this work.

The variables used in the minimisation function are:

q is a component of the processing request, consisting of a service, input parameters and output parameters

s is the service featured in the processing request component q

h is the node to which the service is assigned

c is a heuristic measure of the *goodness* of a placement decision

I the set of input parameters that are needed by a component

i is an input parameter in the set of I. ie $i \in I$

H a set of nodes that satisfies some criteria

$hosts(x, y)$ is a boolean function which returns true if DISCWorld node x hosts service or data y

The result of each step in the placement process is a tuple (q, h) which is unique for a given processing request component q: $\sum_{h' \in H} q \; assigned \; to \; h' = 1$. Thus, each step produces a mapping of a processing request component to a node in the system. The component is only assigned to a single node within the system.

The objective of the placement process is the minimisation of the overall execution time for each component in a given processing request, q. The minimisation function is $\min(placement)$, with *placement* being defined as $(c, h) = cost(q, H)$. This function returns a tuple (c, h) such that c represents the minimum cost for the component processing step and h is the node chosen to host the component. The set of nodes over which assignment is deemed *viable* is the union of the set of nodes that either host the service featured in the component, $hosts(h, s)$, or host the data needed by the component, $hosts(h, i)$. Thus, $H = \{h | h \in \text{known nodes} \wedge (hosts(h, s) \vee hosts(h, i)) \wedge h \neq \emptyset\}$.

The cost of placing a processing request component, q, onto a member of the viable nodes, H is

$$\begin{aligned} cost(q, H) &= \sum_{j \in (I \cup s)} cost_{xfer}(j, H', h) \\ &+ cost_{exec}(s, h) \end{aligned}$$

where $H' = \{h | h \in \text{known nodes} \wedge hosts(h, j) \wedge h \neq \emptyset\}$. We define the cost to transfer a service or data between nodes in the distributed system, $cost_{xfer}$, as $cost_{xfer}(j, H', h) = \min cost_{xfer}(j, h', h) \forall h' \in H'$ where the cost to transfer an object is given below.

$$cost_{xfer}(j, h', h) = \begin{cases} 0 & \text{if } h' = h \\ error & \text{if } h' \neq h \wedge (mobility(j) = 0 \vee \\ & \text{accept new jobs}(h) = 0 \vee \\ & (j = s \wedge \\ & \text{accept new service byte code } (h) \\ & = 0)) \\ (\text{access cost}(h') + latency(h', h) + & \\ size(j)/bandwidth(h', h)) & \\ \times (mobility(j) + & \\ \text{accept new byte code}(h)) & otherwise \end{cases}$$

From the equation above, it can be seen that the cost for initiating data transfer on an object is related to the amount of time that the transfer will take, multiplied by the sum of the object's mobility and the willingness of the destination node to accept new service byte codes. There is no cost associated with the transfer of an object to the same node at which it currently resides. An error condition is raised if the destination node does not accept new service processing requests or if a service is to be transferred and the node does not accept new service byte codes.

Execution schedules are created with a request-centric view of the distributed system, where the scheduling node uses the most up-to-date information that it possesses to generate schedules. In the event that two services are placed onto a node, if the services are unrelated, the execution times of both services are increased by their run time variance; if they are related, they cannot be run together (by definition due to their temporal ordering). We call the increase in time the *co-location penalty*. It is shown in the equation below.

$$exec_{\text{co-location penalty}}(h, s) = \begin{cases} \text{r-t variance(s)} + & \\ \text{r-t variance(s')} & \text{if } hosts(h, s') \wedge unrelated(s, s') \forall s' \\ 0 & otherwise \end{cases}$$

The cost of a given schedule is only loosely based on the time that it should

take to execute. The final cost that the optimisation function reports is generated from estimations of execution time and transfer time for objects, increased by factors commensurate with the source- and destination-nodes' willingness to perform given operations. The sum of the current node's waiting time, the run time of the service, and the co-location factor gives an estimate of the time that the service might take to execute. The node's willingness to accept new service requests is used as a modifier to this estimate; if the node is reluctant to accept new service requests, this translates into a higher cost to execute the service on the chosen node. Therefore, the total cost to execute a service s on a node h is given by the equation below.

$$\begin{aligned} cost_{exec}(s,h) &= (\text{current waiting time to start new services}(h) \\ &+ \text{run time}(s) + exec_{\text{co-location penalty}}(h,s)) \\ &\times \text{accept new job factor}(h) \end{aligned}$$

5.4.2 Schedule Creation and Processor Assignment

The basis of this scheduling model is on clustering. Sarkar [201] also defined clustering as processor assignment in the case of an unbound number of processors on a clique architecture. According to Gerasoulis and Yang's [86] classification, a clustering is called nonlinear if independent tasks are assigned to the same node; linear otherwise. The clustering used in this model is nonlinear.

In order to keep the complexity of the clustering algorithms polynomially bounded, Gerasoulis and Yang only consider those clustering algorithms that do not involve backtracking. In other words, if a cluster is created at step i, then the nodes contained within it are not able to be separated in step $j, j > i$. We also impose this restriction.

The algorithm that we use is similar to the brand-and-bound algorithm [236] in that the first single complete path is found, and its cost is used as an upper bound on the cost. As the remainder of the combinations for placement are tried, if any of the partial paths have a cost greater or equal to the lowest complete-path cost so far, the search is terminated, as the complete-path cost must be greater than the minimum found so far. In the current prototype there is no practical limit on the number of nodes that are considered when creating complete-path costs. In a production system where there may be a large number of nodes that can be considered to host a service or data, it is anticipated that a heuristic would be used to further restrict the nodes considered for placement.

As seen in section 5.3, the DJPL grammar contains loop constructs and aliases. We are unable to mandate that all users must interact with DISCWorld via a graphical interface and automated request-creation tools. The loop construct and aliases are provided to reduce the burden when manually creating processing requests. These constructs exist solely for the benefit of the clients – they are not used by the tools that assign services to nodes and execute processing requests. Alias substitution is performed at the outset of parsing.

The parsing of a DJPL script is recursive. Information present in a given level of nested recursion is available to all lower levels. In fact, if there are any aliases in the DJPL script, they are added as level 0 of the parsing. Instructions are added beginning in level 1.

After any aliases are added to the system, all loops are unraveled. Because we do not allow dependencies between loop iterations, the order of unraveling is not critical. There is no limit as to the number of nested loops. In fact, the current value of the loop control variable is added to the system as a temporary alias; therefore it is undefined outside the scope of the loop.

As discussed in section 5.2.3, one of the fundamental features of this system is a global naming scheme. Because the names of all input and output parameters are explicitly stated in the DJPL script, it is a simple operation to query a local database of available data products and services. If all the outputs of a service are available to the scheduler, the service invocation is replaced with a reference to the data in the store. Because partial results of processing requests are considered to be as important to the system as the final results, if not all of the outputs of a service are available, the service invocation is not replaced.

Each service that is recognised as being necessary to perform is placed according to the scheduling algorithm. A list of candidate nodes is generated, consisting of those nodes that already host the service, and those that host the data which is to be used as input to the service. The use of such a candidate list of restricted nodes is one method by which the complexity of the scheduling algorithm may be reduced.

For each node in the candidate list, the cost of transferring any necessary data and the service is calculated, using the most up-to-date information to the local node. Each of the data items that is needed for the service, as well as the service code itself is now considered. If the data item or service does not currently reside on the node that is being considered for transfer, then the minimum-cost transfer is determined to copy the data or service byte-code to the machine. The minimum is determined

```
1. Let H be the list of nodes that claim to host the service or
   the necessary input data
2. For each location h (h in H)
   2.1 for the service and all data that the service needs .. (x)
       2.1.1 if x is not already hosted at h
         2.1.1.1 find the location and minimum cost to transfer x to h
         2.1.1.2 increment the total cost of the system by the
                 transfer cost
   2.3 add the cost of executing the service on h
   2.4 recursively call step (1) on the next instruction until no
       more instructions
   2.5 if the total cost is less than the current minimum, then
       replace the minimum
3. Return minimum cost found
```

Figure 26: Pseudo-algorithm for determining service and data placement in our scheduling model.

by enumerating over each of the candidate nodes that purports to host the data or service. If the service or data item is already on the node being considered, the incremental cost is zero.

After the cost of transferring any data and the service byte-code to the chosen node is determined, the incremental cost of this configuration is updated and if it is greater than a minimum cost for the complete request, the search is cancelled, and the process is started with the next node to be considered. Otherwise, the algorithm recursively places instructions until there are no more instructions. The pseudo-algorithm is shown in figure 26. The nodes that lead to the lowest overall execution cost for the processing request are chosen.

In our model, if the scheduler does not know of any nodes that host a particular service, then it can create the most efficient schedule it can for all services up to the unknown service, and then start the computation executing. The belief is that when the partial schedule arrives at a remote node with the capability to re-create a schedule, or it arrives at a node where the next service is not assigned to a processor, the new schedule is created, if at all possible. When data or services are referenced in a processing request but are unknown to all nodes participating in the request execution. the behaviour of the system is not defined. We expect to introduce exception-handling instructions into the DJPL grammar in the near future.

5.5 Discussion

Although we are exploring other algorithms, the current method for determining the placement of services and data is by searching the whole state space of those nodes that may be considered to host the service. The algorithm was designed to be recursive to allow additional prospective system state information to be passed to each subsequent instruction that is to be tested. Included in the system state information are details about previous placement decisions that have been made, and the locations of any services or data that have been transferred as part of the placement process. This prevents the placement algorithm from having to pay extra to use a data item that was transferred to the local node in a previous iteration. Using a recursive algorithm also allows the information regarding the remote DISCWorld nodes to be prospectively updated in terms of the services that they host and the expected waiting time that any new service will endure if it uses the node.

While loops were added for the benefit of human readers of the DJPL scripts, they are explicitly expanded due to the fact that it may be more economical, in the presence of a large number of iterations, to transfer some of the data to be used away from the remainder of the processing, in order to achieve a lower waiting time for a service on a node.

As the system under consideration is service-oriented, and does not accept arbitrary user code for execution, is possible, in some ways, to abstract away from the node on which a service is executing. This means that it is not necessary for each server to be aware of the relative speeds of all other nodes that participate in the infrastructure, or of their relative memory sizes as in some other systems [69,95,233].

We do need to record some physical characteristics to help us make decisions on where to execute services, though, such as the current load on a remote node, perhaps in terms of how much of its queue length is full, or perhaps by providing an estimate on the time that an incoming request is likely to start executing. This allows us to break away from the problem of how to properly define a node, especially in the light of having so many different architectures in the processor pool.

Later research may involve the storage of the input graphs and the comparison of derived placement schedules with new graphs, as is detailed in [212]. Predictions are derived from run-times of previous parallel applications, and are used to refine the predictions of current workloads.

It may be possible to use a critical path method (CPM) together with a form of task duplication, such as is used in [205]. Of course, we would have to relax

the constraints that they place on their algorithm, namely that we cannot assume an unbounded number of processors, cannot assume that the program is running in isolation, and the DAG must be able to have more than one root node. In addition, part of the future work that may be performed on the proposed scheduling and placement algorithm, the inclusion of service run-times proportional to the input data, such as detailed in Lee, Yang and Wang [144] will be considered.

5.6 Conclusion

This chapter has introduced two important ideas for the adaptive generation and execution of schedules in a metacomputing context. The first is a theoretical model of scheduling in our prototype metacomputing infrastructure. The second is a prototype language with which computations may be structured and specified in a programming-language independent fashion.

We have presented the mechanism by which processing requests within the DISCWorld prototype are represented. The Distributed Job Placement Language (DJPL) is intended to be automatically generated by a graphical user interface client or a DISCWorld daemon. While users will not be prevented from viewing the generated DJPL script, there should be no need for them to do so.

We believe that the proposed DJPL is sufficiently general that it could be implemented and used in other metacomputing and middleware systems for exactly the same purposes. This language, together with a standardised method for the inter-operation of data and services, would allow a number of different environments to properly cooperate.

Designed for use in systems that feature only partial system state information, the model relies on mechanisms by which information about services and data may be passed about the system. An implementation of this mechanism, and of the model, is presented in chapter 6, and performance analysis of the implementation is provided in chapter 7.

Chapter 6

Implementing Scheduling in DISCWorld

This chapter provides a description and discussion of the experimental framework constructed to implement the adaptive execution schedules as described in chapter 5. We discuss the two fundamental methods by which schedules may be executed, data push and data pull, in section 6.1. Section 6.2 discusses the global naming mechanism that underpins this work. Section 6.3 presents a series of related high-level data abstractions that have been developed for remote data access in distributed object-oriented middleware; section 6.4 extends the abstraction to include data that has not yet been created.

The manner in which the DISCWorld daemon works is described in section 6.5 and processing request execution is described from the global system and local daemon perspectives. An example is discussed, and a summary is presented in section 6.6.

We characterise the DISCWorld execution model as client/multiple-server. This refers to the fact that clients, directly accessible by users, have the majority of the abilities of servers, and that servers can be clients to other servers. For example, a client may have the ability to store data or services, to inject data into the system, and to perform simple processing on data. When servers submit a request to another server or a user client, they temporarily become a client to that server. For simplicity, when there is no need to distinguish between clients and servers, we use the term *servers*.

The majority of non-trivial DISCWorld services are quite complex, taking in the

order of a few minutes of CPU cycles on an average workstation [1] in the processor pool. Because of the time that a daemon invests in the application of services to input data, the results are most likely to be stored for an extended period, or perhaps transferred after a while to a specialised storage server. In addition, the results are likely to be quite large in terms of physical size. Thus, the rationale is that the larger the results are, and the more computational effort that has gone into generating the results, the more such a result should be kept in case it proves useful to a subsequent request.

Data and services can be replicated across nodes in DISCWorld. If a daemon requires some data that resides on a remote node, then a cost calculation is performed to decide whether it is more economically feasible to copy the data from the remote daemon to the local, or to move the actual service code and any other data that is needed by the service, to the remote node. Of course, not all services, and not all data in the DISCWorld system can be copied this way. For example, if a service is implemented as legacy code, for example, in Fortran, it may not be able to be moved [184], and if data is commercially sensitive, only certain operations, pre-defined by the data's owners, may be able to be performed on it.

Another important characteristic of the DISCWorld system is that there are no guarantees that each daemon will have the same view of the global system. Partial system state information is discussed in section 3.4. There are two main tasks that a server is concerned with: maintaining an accurate view of the current system state; and, performing services in order to satisfy queries. As will become clear, maintaining an accurate view of the system state is a useful by-product of a server performing services on behalf of other clients and servers.

6.1 Schedule Execution Models

There are two main approaches to executing a schedule, depending on the amount of system state information available, and whether the programs to be scheduled have been statically assigned to processors. The approaches are data push and data pull. The remainder of this section describes the two models and explains why we have implemented the latter.

[1] Including several DEC Alpha 4/255 workstations, a Sun E250 and a number of Pentium II 266MHz machines running Windows NT and Solaris 2.6

A data push model of execution is characterised by the static specification of the program to be executed. This is does not mean the complete schedule must be statically assigned to processors; it is sufficient for only the program which is to be executed to be assigned. The static specification can be parsed so at each point in the execution, the consumer program which is to use the output of the producer program, is known. This knowledge is used to properly address the data that is to be sent to the consumer program.

As the execution schedule is fixed, there is no need to waste communications bandwidth to ensure the data to be sent is required. This model of execution suits those schedulers that possess complete system state information, because by using the knowledge of the complete system, it is possible to make *known* accurate execution schedules. There is no possibility that other mechanisms which could be used to optimise the schedule will become available. Complete system state information is discussed in section 3.4.

However, in the case in which complete system state information is not available, a mechanism is needed to enable schedule execution to be optimised, if at all possible. The mechanism must enable the optimisation of a schedule, while still providing good execution performance. One method of providing such a mechanism is through the use of data pull execution, described and discussed in the remainder of this section.

Although this is not the original context in which demand-driven, or dataflow models were used [5, 135], the data pull model of execution allows for the dynamic optimisation of execution schedules by the delegation of data transfer decisions to those daemons that require the data. In the data pull model, the machine on which the consuming process is running requests the data from the producing process' machine. Thus this model allows the process to choose where it will retrieve its data from; it does not simply receive its data blindly, as in the data push model.

In order for a consumer process running on a machine to request data from a producer process, it must send a requesting message. Due to this continual exchange of messages, the data pull model places more demands on the interconnection network than the data push model. When comparing the size of messages to the expected size of bulk data, the message traffic becomes insignificant.

If a scheduler has complete system state information, there should be no need to optimise the execution of schedules at run-time, as the information is, by definition, complete. Thus, executing schedules in a data pull model would be unnecessary, as this would only lead to extra message interchange and loss of efficiency. In contrast,

where there is only partial system state information, the daemon that creates the execution schedule cannot be assured that the results that it is about to generate are not already present in the system, but on an unknown machine.

The pull model is more appropriate in the prototype DISCWorld implementation because of the frequent re-use of results, the longevity and size of results, and the fact that in a real system it is not uncommon for machines to fail or become isolated in terms of the network connectivity.

Thus, it is possible that a single piece of data may be found on a number of different DISCWorld nodes, and each daemon may have a different policy by which other nodes may access its data and how much they are charged for the privilege. Because not all daemons share the same view of the global system, it is quite possible that some daemons will be aware of the existence of data that others are not. This means that it may be cheaper for a daemon to retrieve the data from another daemon than that which it was designated to retrieve the data from in the original (static) schedule. If the data distribution model were a push model, then the daemon would have no choice of where its input data came from; in the pull model, the final decision always rests with the daemon that needs the data to perform a service.

Nearly all computing resources, whether they are designed for high-performance computing or storage, are subject to periods of down-time and network vagaries. Due to the non-robust and intermittent nature of the machines that comprise the DISCWorld testbed, the model that has been adopted for request execution in the DISCWorld is that of a *pull* model. When a machine becomes unavailable either due to a system failure, routine maintenance or isolation through the interconnection network, messages will not be able to be delivered to the daemon. In such a case, the method of re-routing the computation such that the failed host must be performed. We believe it is in this situation where the DISCWorld's pull model has the advantage over a push model.

6.2 DISCWorld Global Naming Strategy

One of the fundamental assumptions of this work is the existence of a global naming strategy. We assume there exists a mechanism whereby names can be assigned to data and services in such a way as to identify them throughout the system in a platform-independent manner.

We assume that one of the properties of the global naming scheme is if entities in the DISCWorld share the same name, they refer to the same conceptual object. In this case, the use of the word *object* refers to a high-level object, describing data or service byte-code, divorced from any programming-level semantics. We do not require that the objects are able to be compared in order to assess their equivalence.

In order for two data entities to have the same global name, they must be either pointers to the same *physical* data or pointers to distinct, but *equivalent*, instances of the same data object. The objects may have been created by different byte- or object-code, but if they share the same name, the data is defined to be the same.

When two service entities have the same name, they are defined to accept the same data inputs and produce the same outputs. Furthermore, the outputs that the identically-named services produce are defined to be identical (for the same inputs). For example, a portable version of a service may be available, with the advantage that it runs on any platform, but it may execute slowly or inefficiently. Another version, implemented as highly-optimised, platform specific object-code (eg High-Performance Fortran code [141]), may use parallel libraries and only be available on a small number of nodes. Both versions of the service can be used, with a trade-off of speed for platform independence. Thus, while the actual contents of service entities are not identical, they are *conceptually* identical. The identically-named services are used to allow the system to interchange service implementations without the knowledge of the user.

For the purposes of this thesis, we assume the global name-space is arranged in a hierarchical fashion, and it is possible to derive the *recipe*, or set of instructions, to recreate a derived data item from its name. We name derived data, or the output of a service, according to the service that produced it, the name of the service's output, and the input parameters that were used.

For example, consider a service named `GMS:RetrieveImages`, which retrieves images from an archive corresponding to a particular date and time. The output parameters of the service are the different spectra that the service returns: `IR1`, `IR2`, `IR3`, and `VIS`. Consider the naming strategy for integers as `Integer:xx`, where `xx` is the value. The service invocation corresponding to retrieving the images for 00hrs, 25th December 1998 is `Service:GMS:RetrieveImages(Integer:00,Integer:25,Integer:12,Integer:1998)`. The output data corresponding to the IR1 spectrum image is named `Service:GMS:RetrieveImages:IR1(Integer:00,Integer:25,Integer:12,Integer:1998)`.

It must be pointed out, though, that we *do not* claim to have solved the global name-space problem; nor do we expect the currently-implemented system to be scalable to large numbers of types and services. No attempt is made to provide a general solution to the problem of organising the global name-space; nor are any methods for ensuring type-equivalence. Furthermore, we make no attempt to provide equivalence resolution between branches of a global name-space tree.

6.3 DISCWorld Remote Access Mechanism

Because nodes in the DISCWorld are assumed to be physically distributed, and certain nodes may be owned and maintained by different organisations, we believe it is unreasonable to assume that all necessary data will be available via a common file sharing mechanism such as NFS [216] or DCE [197]. As such, a data transfer mechanism is needed to allow data to be identified and shared amongst many different distributed nodes.

The main objective of the DISCWorld Remote Access Mechanism (DRAM) is to allow client programs to retain a token, or *pointer*, to a remote data entity. The token may be also be used as an input to further processing and allows different application components running on the client to exchange remote data references. DRAMs may either be evaluated at the client side and the data they point to downloaded to the client, or may be used in a subsequent instructions to the server to further process the data, or indeed to pass the data to other servers, which themselves act as clients to the original server. DRAMs act as a proxy for the data to which they refer, but in the context of DISCWorld do not suffer from the dangling pointer problems that a simple implementation would incur.

We first describe the use of DRAMs as pointers to data entities. These data entities may be very large data files which are not directly useful for the user to download to their local computer. Instead the catalogue browser returns a set of references, or DRAMs, to the user that act as pointers to the remote data they have just found. The remote object that a DRAM refers to can either be a data entity or an operation, or service, that embodies some code that can be executed on a DISCWorld node.

The DRAMs that the user now has may be manipulated in two different ways. Within his DISCWorld client environment he may manipulate them *locally*, as graphical icons. More typically the DRAM is just used as a way of specifying the

remote data to be used as input to a subsequent computation. In this case however the data may be relatively lightweight in size and the DRAM representation is encoded to actually carry the data as a cached copy inside it. DRAMs therefore provide a useful way to encode objects for transmission between hosts depending upon whether they represent large or small data sets.

Typically, a remote pointer is an object which records the name of the remote host and the location in memory of the object being referenced [73]. The dangling pointer problem occurs when the remote object that is referenced is no longer available when the pointer is dereferenced. The most common cause of dangling pointers is when the server program crashes or is restarted – the data objects that are created on the server side will, in all likelihood, be stored at different locations in memory. Thus, any existing remote pointers may reference unused portions of the server's memory or at worst, different objects.

As all data and services (or objects) within the DISCWorld system are referenced by *global name* rather than *daemon memory location*, we can circumvent the dangling pointer problem for DRAMs. When a daemon receives or creates an object, it is stored in non-volatile memory before a DRAM is able to be created to point to it. Thus, the existence of a DRAM referencing an object on a remote daemon implies that the remote object is in non-volatile memory. This condition guarantees that when a DRAM is inspected to access the remote data, the data will be on the remote daemon. In addition, the nature of the data products that DISCWorld manipulates is such that the DRAM can carry a descriptive recipe of how to reconstruct the data product even if the data is no longer stored where the DRAM originally pointed to.

In the DISCWorld computational model, remote pointers or DRAMs are sent between clients and servers, and servers and servers. These provide a convenient and powerful mechanism for the propagation of result- and service-metadata throughout the DISCWorld. Because data and services in the DISCWorld have unique, global names [111], results of the same service invocation (with the same input parameters) created on different DISCWorld nodes can be easily tested for equality.

We use the following notation when describing the specification and behaviour of DRAMs:

- d represents a DRAM,
- D represents data,
- S represents a server (or client).

This is shown in figure 27, upon which we base the following description of DRAMs.

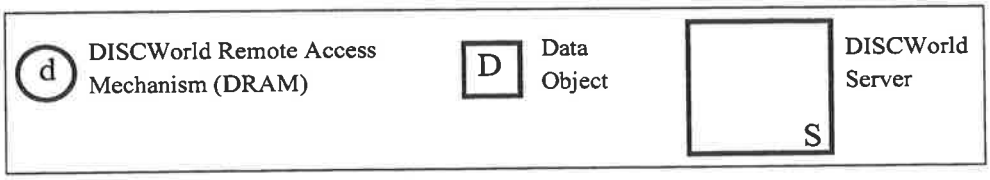


Figure 27: Symbols used to explain the DISCWorld Remote Access Mechanism (DRAM) notation.

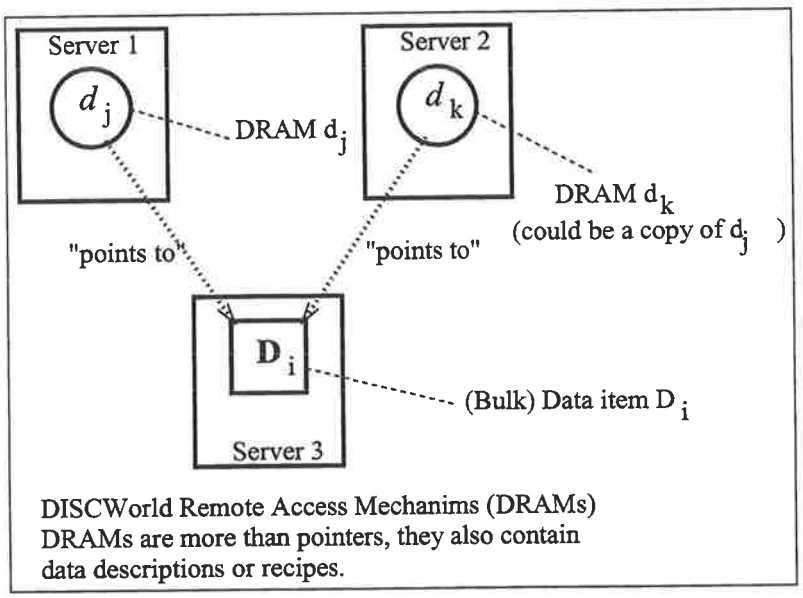


Figure 28: DRAMs allow servers to point to remotely stored bulk data items, and to copy these pseudo pointers to other servers. DRAMs are more than pointers, since they contain a description of how a DISCWorld server could (re)construct the bulk data item if necessary.

DRAMs are defined to point to data, not other DRAMs. This therefore limits the number of indirection levels to one and reflects our design goal that DRAMs be pointers to either bulk data products or code services. The notation of $d_i \rightarrow d_{i+1} \rightarrow D$ signifies that both d_i and d_{i+1} point to D, not each other, and that one may have been created by copying, or cloning, the other.

$$d_{\{i\}} \rightarrow D_j \text{ means } d_{\{i\}} \text{ each point to } D_j \forall i$$

If the data, D, that a DRAM, d, points to is discarded by the server, by virtue of the scheme whereby the server's name for the data uniquely describes how to make it, the data can be recreated by the server if it is not able to be located using a resource discovery mechanism.

$$d_i \rightarrow d_j \rightarrow D_k => d_i = d_j$$

It is possible to have DAGs of d_i's. There is an $n : 1$ relationship between d and D.

$$\begin{aligned} d_1 \rightarrow d_2 \rightarrow \cdots \rightarrow d_j \rightarrow \cdots \rightarrow d_n \rightarrow D_x \quad & => d_1 \rightarrow D_x \\ & => d_2 \rightarrow D_x \\ & \vdots \\ & => d_n \rightarrow D_x \end{aligned}$$

The most important DRAM concept is embodied in figure 28. A set of three DISCWorld nodes, servers as shown, cooperate in some calculation. Server 1 has a DRAM d_j that points to a bulk data entity D_i located under the control of Server 3. Server 2 has been given a DRAM that also points to this data entity and it may have either been set up as an independent transaction between servers 2 and 3 or it may have been passed a copy of Server 1's d_j. In either case both can refer to the bulk data entity and can use it as input to remote computations. The data belongs to Server 3 and only Server 3 can set the actual policy for deletion of the data entity. Either or both of servers 1 and 2 can delete their DRAM reference to it however. Typically the data entity will be a long lived data product that may be archived to a long term store according to Server 3's policy specifications, but is unlikely to be deleted outright. Alternatively it may be a derived data product that Server 3 knows how to recreate locally should it decide it is more cost effective to do so rather than store it.

The Java class definition of a DRAM is shown in figure 29. We have specialised the concept of a DRAM to be able to point to both services and data. These are named DRAMSs and DRAMDs respectively, and their interfaces are shown in figures 30 and 31. Examples of their contents are shown in figures 32 and 33. Figure 32 shows a high-level service representation of a service found in the ERIC application; figure 33 shows the result of using the DRAMS to produce some output data. DRAMs are explained more fully in [111].

```
public abstract class DRAM implements Serializable
{
  private String publicName;     // descriptive name for GUI use
  private String globalName;     // internal ID
  private Icon icon;             // associated icon eg thumbnail image
  private String description;    // long free textual description
  private String className;      // query-able search-able text
                                 //   representation of class
  private String remoteServer;   // location of the Data to which
                                 //   the DRAM points
  private int objectMobility;    // whether the object being pointed to
                                 // can be transferred across
                                 // the network
  private int objectSize;        // size of the object being pointed
                                 // to, in bytes
}
```

Figure 29: DRAM Java base class. The DRAM provides a high-level pointer to an object.

6.3.1 Operations on DRAMs

There are a number of operations that can be defined for DRAMs. We summarise these symbolically in table 7. Figure 34 shows the consequences of DRAM creation, copying and moving. These are the most important operations.

Create This operation is the most fundamental operation that can be performed on a DRAM. Once the DRAM object is instantiated, the act of invoking its create method causes it to be bound to a data object, as shown in figure 34 i). This will typically be done as one operation using the DRAM constructor.

```
public class DRAMS extends DRAM implements Serializable {

  private DRAMP []inputParameters;  // parameters for the service
  private DRAMP []outputParameters;
  private Class localClass;         // Byte-code for the portable
                                    // service
  private int runTime;              // mean number of seconds this
                                    // service executes for on the
                                    // source node
  private double runVariance;       // variance of execution time from
                                    // mean
}
```

Figure 30: Service DRAM (DRAMS) Java class, which extends DRAM. The DRAMS provides a high-level representation of a service, which may be more easily transferred between hosts than the byte-code. The DRAMS can be used to make decisions on where to execute the service.

```
public class DRAMD extends DRAM implements Serializable
{
  private Object localObject;   // the local (cached) copy (if any)
  private DRAMS []operations;   // allowable operations on the object
}
```

Figure 31: Data DRAM (DRAMD) Java class, which extends DRAM. The DRAMD provides a high-level representation of data. It may be used to make decisions on the best place to send the data for use in a computation.

Operation	Definition
create	CR:: $D \rightarrow (D, d)$
copy	CP:: $d \rightarrow (d, d')$
move	MV:: $d_s \rightarrow d_{s'}$
copy_data	COPY_DATA:: $(d_s, D_{s'}) \rightarrow (d_s, D_s, D_{s'})$
capture_data	CAPTURE_DATA:: $(d_s, D_{s'}) \rightarrow (d_s, D_s)$
discard	DISCARD:: $d_s \rightarrow \emptyset$
inspect	INSPECT:: $(d_s, D_s) \rightarrow (d_s, D_s)$

Table 7: Formal definitions of operations on DRAMs, where each definition is in the form of before→after and s, s' are servers

Instance Variable	Typical Value
`publicName`	ERIC Retrieve Single Image Service
`globalName`	Service:ERIC:SingleImage
`icon`	*assigned by creator of service*
`description`	ERIC Service for Retrieving Single Image from GMS5 Satellite Repository
`className`	satellite.GMS5.ERIC.SingleImage
`remoteServer`	cairngorm.cs.adelaide.edu.au:1965
`objectMobility`	2
`objectSize`	3232
`inputParameters`	*array containing parameter structures to represent integers for the day, month, year, time and scale of image to be returned, as well as the area of interest, and a string representing the spectral channel*
`outputParameters`	*array containing a structures of type image.satellite.GMS5*
`localClass`	*null*
`runTime`	128
`runVariance`	14.3

Figure 32: Example contents of a DRAMS. This DRAMS is used in the ERIC application (see appendix B), and retrieves a single image from the satellite imagery repository. The DRAMS originates from the node `cairngorm.cs.adelaide.edu.au`. As the mobility of the service is 2, it can be transfered around the system.

Copy Copying a DRAM causes its contents to be copied into another instance of the DRAM object. This operation is necessary to allow the propagation of DRAMs around the DISCWorld system when a daemon wishes to retain a pointer to the remote data. This is shown graphically in figure 34 ii).

Move This operation provides the mechanism for a DRAM to move between servers in the DISCWorld system, as is illustrated in figure 34 iii). It is by this mechanism that DRAMs are transferred between DISCWorld servers. This operation causes the DRAM to be serialized, encapsulated in a message and sent to the destination server.

Copy_Data This operation causes a DRAM to download a copy of the data to which it refers from the remote server. This is shown in figure 34 iv). Copy_Data is

Instance Variable	Value
`publicName`	GMS5 1998122500:IR1 (0,0),(2291,2291) 100% zoom
`globalName`	ERIC:SingleImage:GMSImage:IR1(Integer:00,Integer:25, Integer:12,Integer:1998,Integer:0,Integer:2291, Integer:0,Integer:2291,Integer:100)
`icon`	*thumbnailed representation of data*
`description`	GMS5 image 00Hrs 25DEC1998, IR1 channel, Original Dimensions, 100% zoom
`className`	image.satellite.GMS5
`remoteServer`	cairngorm.cs.adelaide.edu.au:1965
`objectMobility`	2
`objectSize`	5248681
`localObject`	*null*
`operations`	*array containing references to crop, histogram equalise and image convolve operations*

Figure 33: Example contents of a DRAMD. This DRAMD is the result of an invocation of the DRAMS shown in figure 32. It originates from the host `cairngorm.cs.adelaide.edu.au`, and the original data object has size 5248681 bytes. The original data object is not bundled with this DRAMD.

most often used by clients before inspecting data, and by servers to create a local copy before using it in a computation. Copy_Data may also be used by those servers that have been designated as archiving machines, for the long-term storage of data.

Capture_Data Capturing the data that a DRAM refers to is the equivalent of moving the data from the remote server to the local node. The node effectively now owns the data entity and no longer retains a reference to it on the remote server. However, it is a matter of policy for the remote sever to decide whether it will actually delete its retained copy of the original data entity.

Discard Causes the DRAM to be completely destroyed. Note that this does not affect the original data entity, local or remote, to which the DRAM points, as data is stored and managed independently.

Inspect Inspecting a DRAM causes the local data to be displayed in the user's workspace using whatever default method is defined for the data type. For example, in the case of a DRAM representing an image, the default action

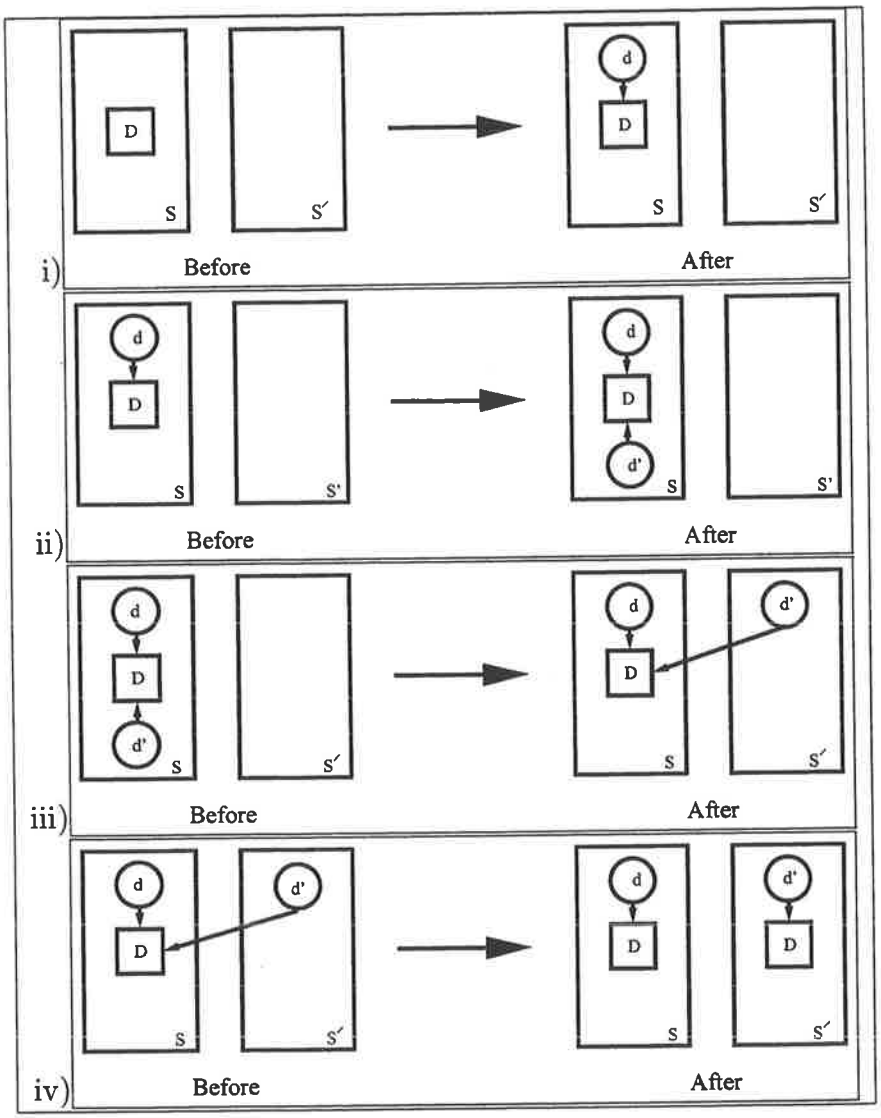


Figure 34: DRAM operations: i) Creating a DRAM, ii) Copying a DRAM, iii) Moving a DRAM, and iv) Copying a DRAM's data before inspecting.

would be to display it in a simple image viewer applet, whereas if the DRAM refers to a byte-code, the service's GUI would be displayed on the workspace.

It is part of the DRAM creation process to instantiate the DRAM reference locally before passing it to a remote node. It is therefore possible for a node to have a number of DRAMs pointing to its own local data.

We rely largely on constraint properties of the bulk data entities (long lived-ness and read-mostly) managed under DISCWorld as well as explicit data management policies set by server administrators to avoid the need for incorporating call-back into DRAMs. A server issuing DRAMs to its data does not retain knowledge, under our present implementation, of where the DRAMs went.

6.4 DRAM Futures

To implement the optimisation of process networks, we have extended the DRAM concept to encompass that of pointers to data that have not yet been created. The extended DRAMs can be passed between clients and servers. We term these pointers to not-yet-created data DRAM Futures, or DRAMFs.

DRAMFs are created with the name of the data that they *will* represent, and as such, are equivalent to DRAMs in the operations that can be performed on them. DRAMFs are assigned an approximate size for the data they point to, based on the mean historical size of the previous data products of the creating service on that node. However, unlike DRAMs, we restrict DRAMFs to only be able to point to data, not services. This restriction is common-sense, as it is not sensible to have a pointer to a service which does not yet exist. As for any DRAM, the `copy_data` operation is used to retrieve the data to which the DRAM points. In the case of a DRAMF, a request is sent to the server that is to create the data; when the data has been created, a DRAMD of the same name as the DRAMF is returned with the data. Thus, the DRAMF is replaced with a DRAMD, which contains the exact size of the data item, instead of the DRAMF's estimate.

The implementation of DRAMFs extends the DRAM Java base class. This is shown in figure 35, and an example of a DRAMF's contents is shown in figure 36. For simplicity, the `estimatedTimeAvailable` field is used as an offset from the time that the holder of the DRAMF requests the data be created. This value consists of the current waiting time for the host node start the service, as well as the mean run time of the service that will create the data. If the data represented by the DRAMF

uses any other DRAMFs as inputs to its service, the `estimatedTimeAvailable` field also includes the estimated time of the DRAMF upon which this service needs to wait, and the estimated time to transfer the resulting data between DISCWorld daemons. We consider the time taken to estimate the data's arrival time to be small when compared with the time taken to transfer necessary data and perform the services that the DRAMF represents.

In principle, DRAM Futures are similar to programming-language level Futures [231], Wait-by-necessity [33] and Promises [152] mechanisms. Whereas Futures and Promises were designed to hide the latency in RPC-based systems, DRAMFs are intended to be used as a high-level pointer to data, in exactly the same way as DRAMDs and DRAMSs. The main difference between DRAMFs and the other mechanisms is their granularity, and the operations which can be performed on them. The Futures and Promises mechanisms are fine-grained. Because the Futures actually represent a memory location in the requesting program, they are tied to the instance of that program. They are not able to be sent between programs. DRAMFs are higher granularity than the other systems, abstracting away from any instance of a creating program.

DRAMFs can be sent between servers, and may also be used by clients in the composition of new processing requests. What this means is that a DRAMF may be copied and moved to many other servers. Subsequently, it may be used as an input to a service by a scheduler, and only when that schedule is *executed*, will the data to which that DRAMF points be copied to a remote machine. Thus, by using the DRAM Futures mechanism, and more generally, the DRAM mechanism, unnecessary bulk data transfers are prevented.

The DRAMF mechanism is used to best effect when executing user requests. The next section explains the method by which schedules are executed in the DISCWorld prototype, and how DRAMFs enable adaptive scheduling across DISCWorld servers.

6.5 Execution in DISCWorld

When a client submits a processing request to its local DISCWorld daemon, an initial execution schedule is created. The local DISCWorld daemon uses its most up-to-date system state information to create a schedule with a minimum execution time. The model and information required for the creation of schedules is discussed in chapter 5.

```
public class DRAMF extends DRAM implements Serializable {

  private int estimatedTimeAvailable;  // estimation of when data will
                                       // be available from the time that
                                       // this DRAMF is dereferenced,
                                       // without any execution
                                       // optimisations
}
```

Figure 35: Future DRAM (DRAMF) Java class, which extends DRAM.

The request is expressed in our Distributed Job Placement Language, (DJPL). The DJPL is discussed in chapter 5.

When executing schedules, DISCWorld daemons do not explicitly order services for execution. Services are executed as the data they require becomes available. In this way, execution in the DISCWorld is very similar to a macro-dataflow model [126]. This section describes the way in which DRAMs are used for both the creation of execution schedules, and the execution of the schedules. The implementation of global scheduling and local scheduling are discussed, and an example is provided.

The DISCWorld prototype is written completely in the Java [88] language. This decision was made because of Java's object-oriented, strong typing, network aware, and platform independent implementation. We believe the security and ubiquity provided by this language far outweighs the speed sacrifice in using an interpreted language. The implementation is slightly in excess of 7000 lines of Java. While all processing routines were hand-crafted, the scanner for the DJPL is a publicly-available XML parser [124]. Although this work is part of a larger project involving many researchers, the remaining subsections describe the actual implementation, not the theoretical model of the system.

Except where explicitly mentioned, the term `DRAMD` should be read as `DRAMD` *or* `DRAMF`. Future DRAMs are equivalent to DRAMDs with the exception that the data to which they point has not yet been created. As mentioned in section 6.4, DRAMFs support exactly the same operations as DRAMDs. The difference between them is that there is a higher latency when retrieving the data to which a DRAMF points.

Instance Variable	Value
`publicName`	Future for GMS5 1998122500:IR1 (0,0), (2291,2291) 100% zoom
`globalName`	ERIC:SingleImage:GMSImage:IR1(Integer:00,Integer:25, Integer:12,Integer:1998,Integer:0,Integer:2291, Integer:0,Integer:2291,Integer:100)
`icon`	*thumbnailed representation of data*
`description`	Future GMS5 image 00Hrs 25DEC1998 IR1 channel, Original Dimensions, 100% zoom
`className`	image.satellite.GMS5
`remoteServer`	cairngorm.cs.adelaide.edu.au:1965
`objectMobility`	2
`objectSize`	5248681
`estimatedTimeAvailable`	102

Figure 36: Example contents of a DRAMF. This DRAMF represents the DRAMD shown in figure 36 before its data has been created.

6.5.1 Schedule Creation

Upon a client being started by a user, the client can request the available DRAMs from its local DISCWorld daemon. In this way, the user at the client can use the DRAMs for composing processing requests. If, in the course of execution, the client receives DRAMs, they are cached. The local DISCWorld daemon is used as an interface into the DISCWorld environment; all processing requests are sent here for scheduling. If a client has a DRAM which originates from a remote server, they can send a message directly to the remote server; there is no need to go through the local daemon like a proxy.

When a processing request is sent from a client to a local DISCWorld daemon, it is expressed in the DJPL in a form which is called *un-annotated*. This indicates that the services which are to be run have not been assigned to servers. This may also mean that any loops in the system have not yet been expanded.

Upon parsing the processing request, which is written in XML, any aliases are substituted into the instruction stream, and any loops in the DAG are expanded. While DAGs are acyclic by definition, in this context we consider loops to be a convenient method for executing the same set of instructions over a *constrained* set of values. Further, we constrain the behaviour of loops to expressly forbid dependencies between iterations. Examination of the syntax (see section 5.3) will show that it is not

easy to construct a loop with dependencies between iterations. In fact, the current lack of a programming language-style syntax makes it impossible to define aliases within the instruction (and hence loop) sections of the DJPL in order to aide in the construction of dependent loops. If a loop *is* specified within the DJPL, the iteration with the backwards dependency is treated as though it appears outside of the loop. When the code is placed onto DISCWorld nodes, and executed, the iteration with dependencies will have the starting time delayed, by necessity. Again, due to the lack of a programming language-style syntax, no consideration has yet been made as to the execution of infinite loops or loops with dependencies between all iterations. The current parser fails when presented with loops that contain dependencies between iterations; the handling of loops with inter-dependencies is an area for future research.

The local DISCWorld daemon examines each of the services and data items used in the request. For every service, the daemon checks that it has at least one DRAMS to that service; for all data items that are used as inputs, the daemon must have at least one DRAMD to the data. The only way a client can use a DRAMD is by actually having a reference. If the daemon does not have an instance of the DRAMD, a request can be made to the client for a copy. Using the algorithm and the cost function described in chapter 5, a heuristically good schedule is created.

The resulting execution schedule is again expressed in DJPL. Because all the services contained within the request have been assigned to servers, the script is termed *annotated.* The annotated DJPL script is sent to each of the remote DISCWorld daemons that have been selected to participate.

6.5.2 Global Execution

This subsection describes the way in which scheduling in the DISCWorld is implemented, from a global perspective. It describes the behaviour of the prototype system from a high-level viewpoint.

As detailed in section 6.5.1, the DISCWorld daemon local to the client from which the processing request is submitted, annotates the request with placement information. The annotated request is then distributed to each of the DISCWorld daemons that are to participate in the execution of that request. Through inspection of the annotated DJPL script, the daemon that creates the execution schedule can determine which DRAMs must be transferred to remote daemons in order for the computation to begin. Only those DRAMs that correspond to data (and services) which exist before the start of the computation are transferred. Thus, by its very

nature, the data that will be created as a product of the computation has no associated pointers. If such pointers already existed, the processing request would be optimised as described in section 5.4.2.

When a DISCWorld daemon receives a DJPL script, it is parsed and if any of the services have not been placed by the daemon that annotated the script, it is updated and the remainder of the daemons are informed of the change. Although it is not expected to be a common occurrence, the daemon has the right to refuse to execute any number of services that it has been assigned.

The best way in which to explain the implementation of scheduling from a global perspective is with an example. Consider the state of the DISCWorld system as shown in figure 37. The figure shows four DISCWorld daemons: the local daemon, L, and three remote daemons, R, R_1, R_2. Daemons R and R_2 host data, D_1 and D_2, while R_1 hosts a service, S. Each of the data and services have associated DRAMs, d_{d_1}, d_{d_2}, d_s respectively. A further example, with timing measurements, is presented in section 7.2.

If a processing request, shown in figure 38, is injected into the system by a client, C, which is on the same node as L, then figure 39 shows an annotated version of the same DJPL script. Figures 40, 41 and 42 show the steps by which the result is computed and returned to the user. The steps are itemised, and explained below. In this example, we assume that the placement algorithm places the service onto server R due to the size of the data that it hosts (D_1).

The first three steps shown below are not strictly part of the execution of a processing query. They are, however, vital steps in the process, and deserve to be discussed in order to maintain the correct sequence of events.

1. **resource discovery.**

   The first step in the execution process involves the creation of a snapshot of the global system state. The local daemon, L sends requests for status updates from all the nodes that it is aware of. If, however, it is aware of a large number of nodes, depending on the daemon's policies, it may only query the nodes which it uses most often, or from which it has not had an update for the longest amount of time.

   System state consists of the components mentioned in section 5.2.3: nodes, interconnection network statistics, and service and data descriptions. Service and data descriptions are in the form of DRAMSs and DRAMDs respectively.

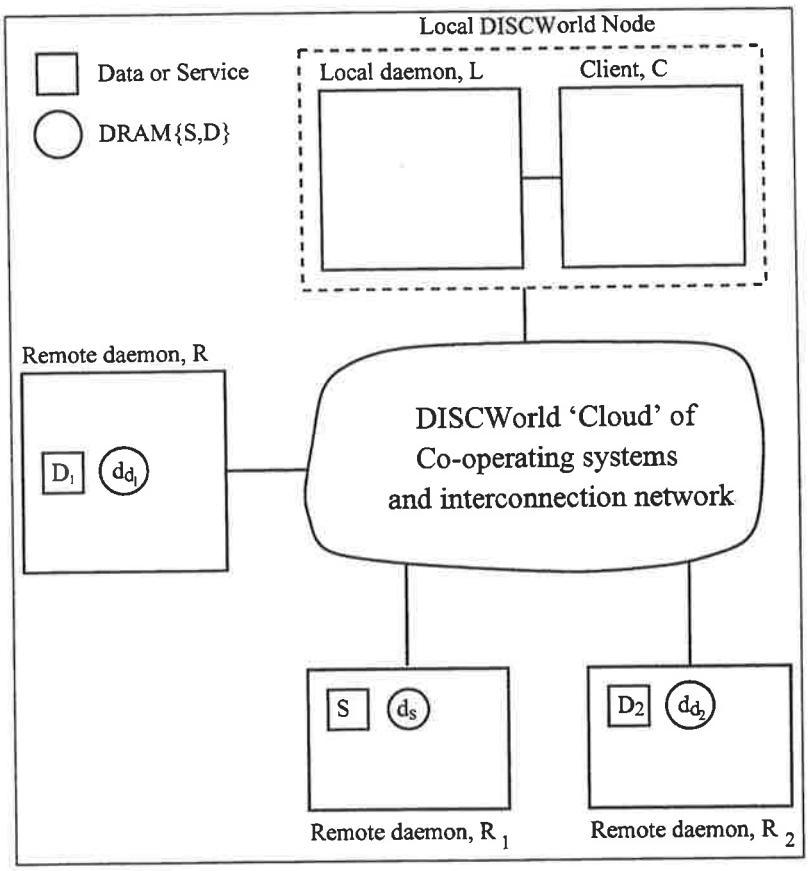


Figure 37: State of the DISCWorld environment before execution of the example shown in figure 38.

As discussed in section 6.3, DRAMs can be traded between DISCWorld daemons without transferring the data to which they refer.

2. **user logs on to client.**

The user client has the ability to store the DRAMs that have been received during the course of computation. Between sessions, the DRAMs are saved on non-volatile media. When the user logs on to the client, the DRAMs that the daemon has available are transferred to the client. These may augment or replace the DRAMs that have been stored by the client.

The current implementation requires users to be logged into the system to submit processing requests and retrieve results. However, the general model

```
<?xml version="1.0"?>
<!DOCTYPE DJPL SYSTEM "DJPL.dtd">
<DJPL>
  <USER>
    <ID>heath</ID><GROUP>dgis</GROUP>
    <PERMISSION>unrestricted</PERMISSION>
  </USER>
  <JOB>
    <ID>00005</ID><PRIORITY>5</PRIORITY>
    <COST><SOFT>100</SOFT><MAX>120</MAX><ESTIMATE>90</ESTIMATE>
    </COST>
    <TIME>360</TIME>
    <SERVER><NAME>C</NAME><PORT>6668</PORT></SERVER>
  </JOB>
  <INSTRUCTIONS>
    <INSTRUCTION> <SERVICE>S</SERVICE>
      <PARAMETER TYPE="INPUT">
        <NAME>input_one</NAME><VALUE>D1</VALUE>
      </PARAMETER>
      <PARAMETER TYPE="INPUT">
        <NAME>input_two</NAME><VALUE>D2</VALUE>
      </PARAMETER>
      <PARAMETER TYPE="OUTPUT">
        <NAME>output</NAME><VALUE>F</VALUE>
      </PARAMETER>
    </INSTRUCTION>
  </INSTRUCTIONS>
</DJPL>
```

Figure 38: Example *un-annotated* DJPL code produced from a processing request. For the purposes of example, the names of the service and the data have been greatly simplified. The method of naming such entities used in the implemented prototype is discussed in section 6.2.

```
<?xml version="1.0"?>
<!DOCTYPE DJPL SYSTEM "DJPL.dtd">
<DJPL>
  <USER>
    <ID>heath</ID><GROUP>dgis</GROUP>
    <PERMISSION>unrestricted</PERMISSION>
  </USER>
  <JOB>
    <ID>00005</ID><PRIORITY>5</PRIORITY>
    <COST><SOFT>100</SOFT><MAX>120</MAX><ESTIMATE>90</ESTIMATE>
    </COST>
    <TIME>360</TIME>
    <SERVER><NAME>C</NAME><PORT>6668</PORT></SERVER>
  </JOB>
  <INSTRUCTIONS>
    <INSTRUCTION> <SERVICE>S</SERVICE>
      <PARAMETER TYPE="INPUT">
        <NAME>input_one</NAME><VALUE>D1</VALUE>
      </PARAMETER>
      <PARAMETER TYPE="INPUT">
        <NAME>input_two</NAME><VALUE>D2</VALUE>
      </PARAMETER>
      <PARAMETER TYPE="OUTPUT">
        <NAME>output</NAME><VALUE>F</VALUE>
      </PARAMETER>

      <NODE>R</NODE>
    </INSTRUCTION>
  </INSTRUCTIONS>
</DJPL>
```

Figure 39: Example *annotated* DJPL code produced from the example shown in figure 38. The service S has been assigned to execute on remote daemon R. The difference between the unannotated and annotated script is shown in bold font.

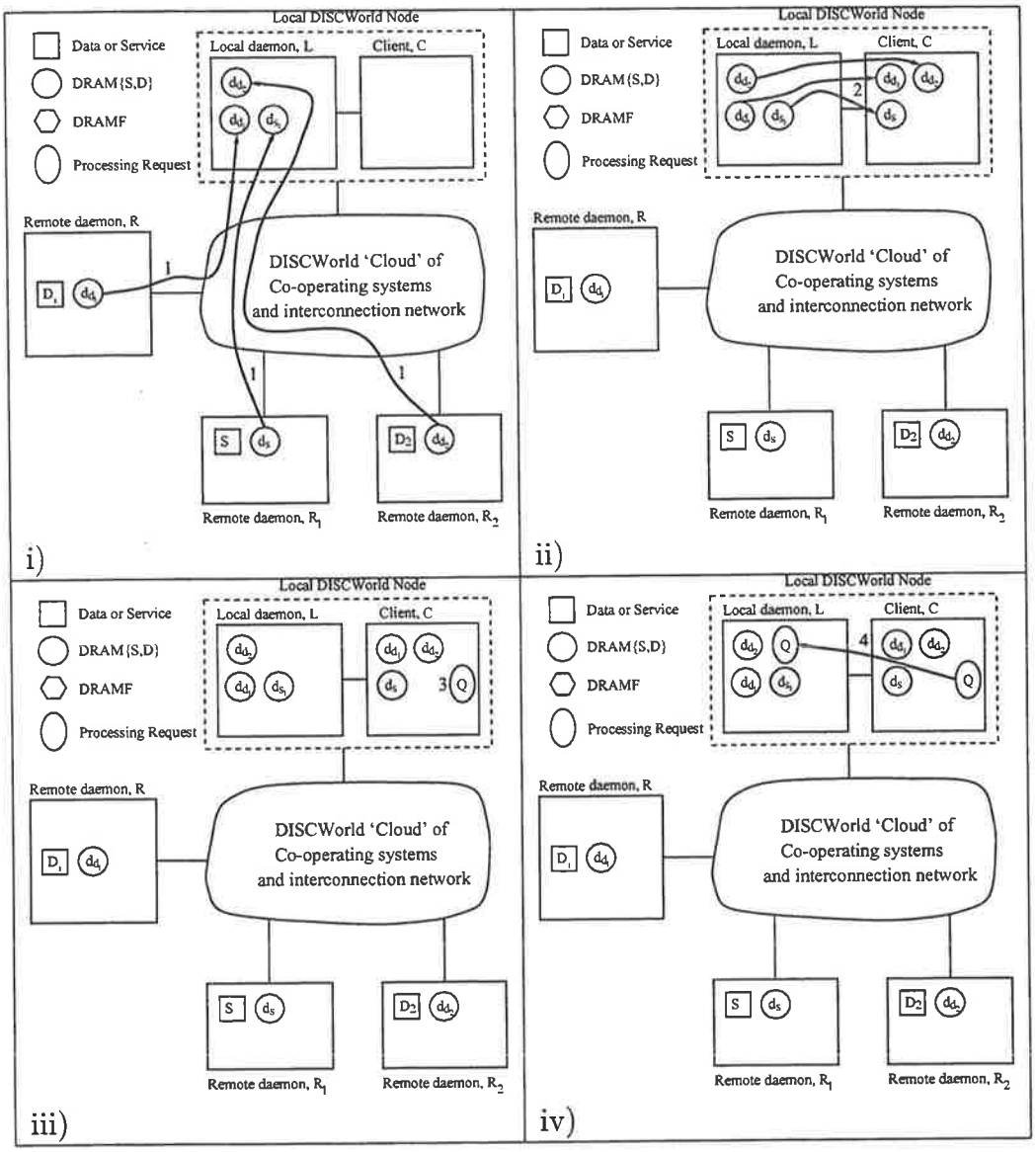


Figure 40: Steps involved in executing a DISCWorld processing request: i) resource discovery, ii) user logs on to client, iii) user composes processing request, iv) request transferred from client to local DISCWorld daemon.

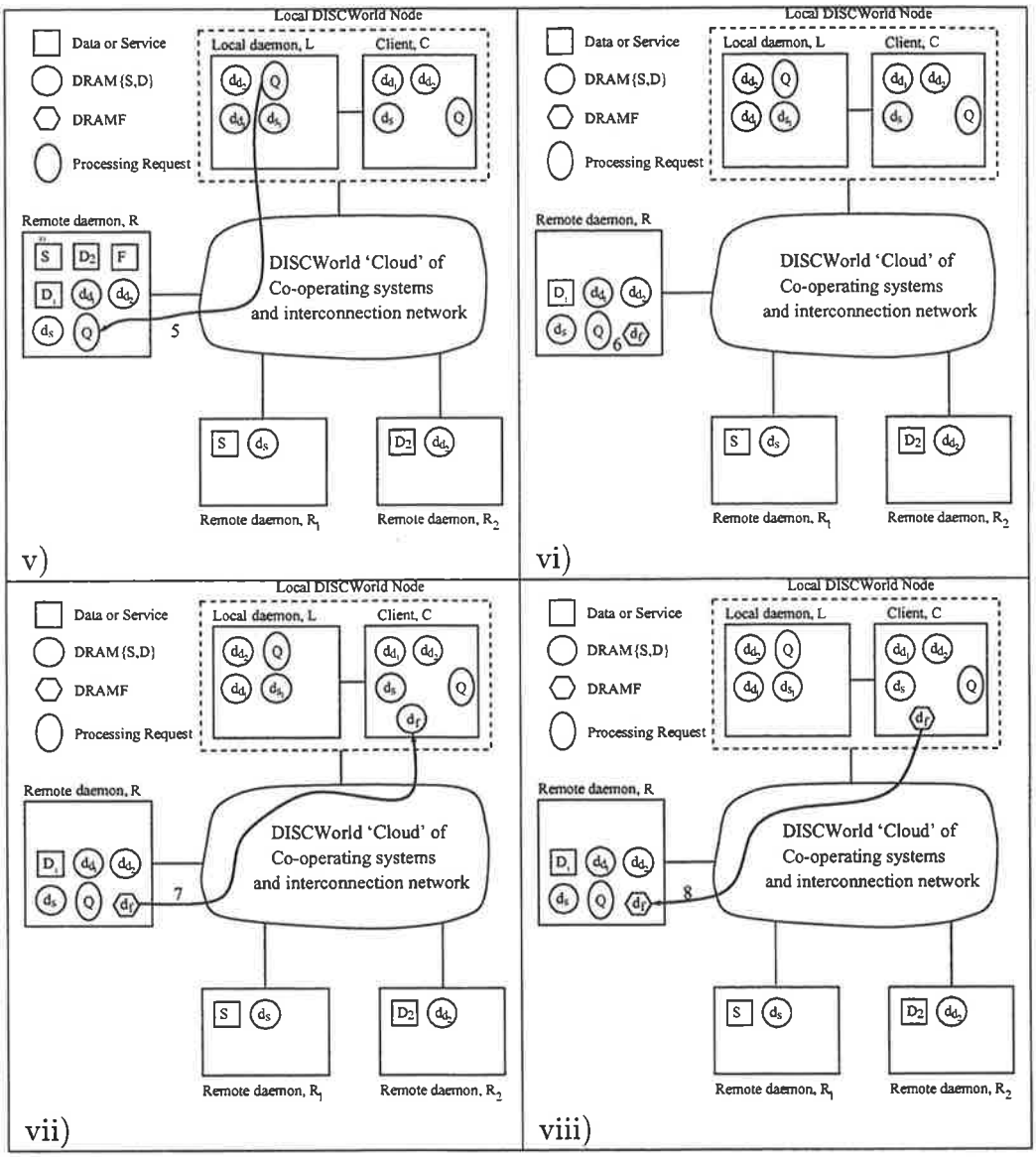


Figure 41: Steps involved in executing a DISCWorld processing request: v) local daemon executes placement algorithm and sends the annotated processing request to the chosen server, with DRAMs to all the services and data to be used on that daemon, vi) the server parses the annotated processing request and creates DRAMF for result, vii) the server sends the DRAMF to the client, viii) the client inspects the DRAMF to begin execution.

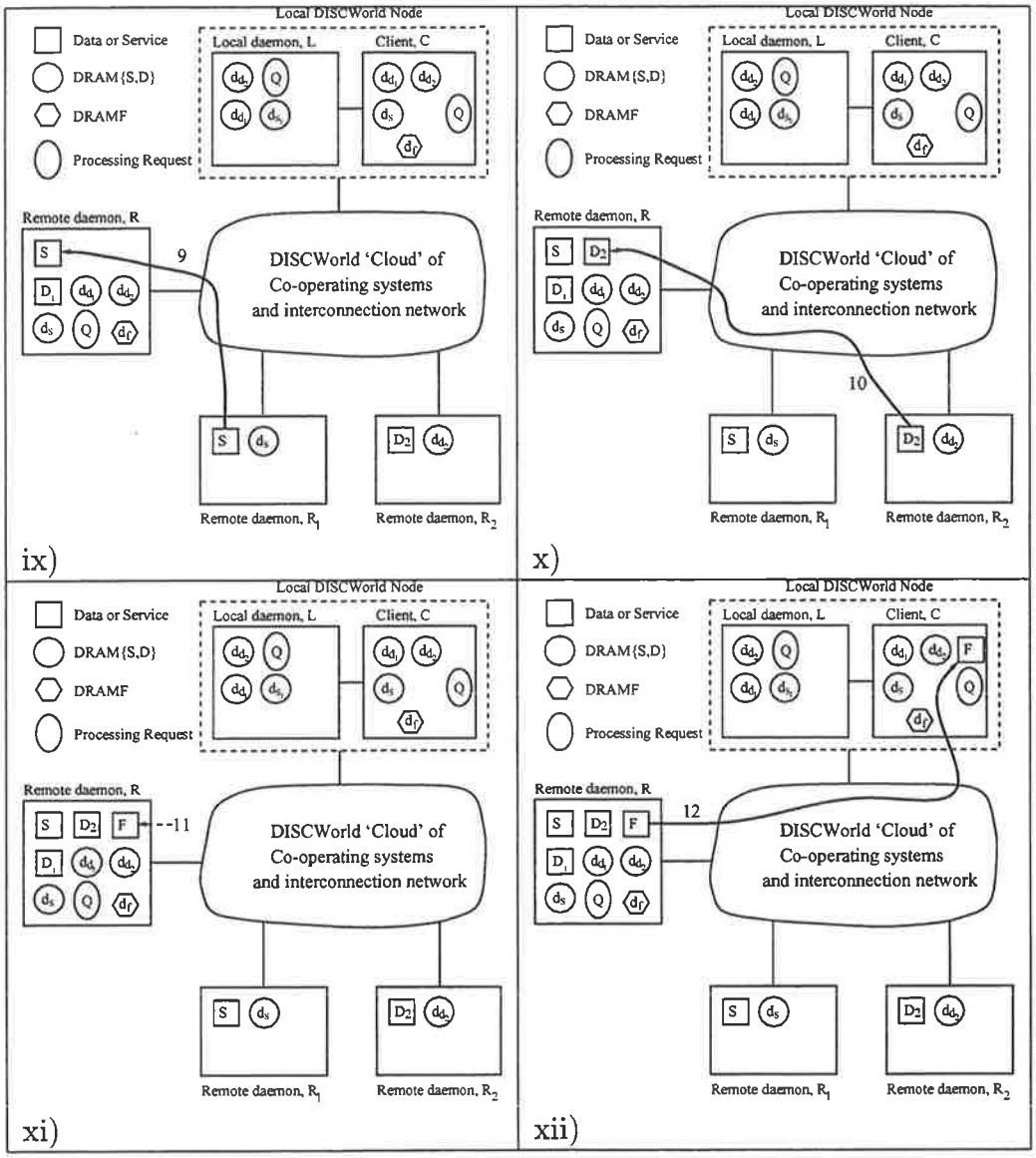


Figure 42: Steps involved in executing a DISCWorld processing request: ix) the server inspects the DRAMS, downloading the service byte-code if needed, x) the server inspects all necessary DRAMDs, downloading the data if needed, xi) the service is executed and the result is produced, xii) the server returns the outstanding request for the DRAMF to the client.

for the DISCWorld system is to allow users to log in from anywhere in the distributed system and be able to retrieve their results.

3. **user composes processing request, Q.**

   The user can create processing requests by combining services and data DRAMs, or by using services locally. For example, a service may present a graphical user interface (GUI) which can run inside the user's client. The user may be able to use the GUI to supply values which will be used in a further query. When the user has completed creating a processing request, they then submit the request to their local DISCWorld daemon.

4. **Q transferred from client, C, to local DISCWorld daemon, L.**

   The request is expressed as a DJPL script and is sent in a serialized form to the DISCWorld daemon, encapsulated in a wrapper which provides the source and destination machine addresses, and limited metadata. The DJPL script is shown in figure 38.

5. **L executes the placement algorithm and sends the annotated processing request (shown in figure 39) to the chosen server R, with DRAMs to all the services and data to be used on that daemon** (d_s, d_{d_1}, d_{d_2})

   After the processing request has been parsed and placed in accordance with the cost functions and algorithm specified in section 5.4, the annotated script is sent to each of the nodes that has been chosen to participate in the computation. In addition, if the services that have been placed onto remote nodes require DRAMs to either services or data, they are sent with the DJPL script. In the current implementation, parsing and placement is terminated if a daemon does not know about any nodes that host a given service, or data. However, the general DISCWorld model is that the DJPL script will be partially annotated. The partially-annotated script is then sent to the first non-local node that has been assigned a service, in the hope that it will know about the service or data that the original did not.

   Only data that exists before the execution of the processing request can be transferred to the remote nodes. Of course, data that is created during the execution of the processing request is not able to be sent out before the request starts executing! If the remote daemon receives a DRAM to data or a service

that it already hosts, then the DRAM is cached, and the remote daemon reserves the right to use any one of its available DRAMs.

Once the daemon that creates the schedule, L, has distributed the annotated DJPL script to all the remote nodes that are to be used, processing ceases unless the daemon has been assigned to perform a service. Although not implemented in the prototype, the DISCWorld model allows the client to query the local daemon as to the progress of the computation. The local daemon can act as a broker, forwarding the request for information to the daemon assigned to perform the service.

6. **R parses Q and creates DRAMF, d_f for result.**

When a daemon receives an annotated DJPL script from a remote node, it parses the script to check that the annotation is complete. If not, placement is completed, and the annotated DJPL script is distributed as per the previous processing step. Once the script has been parsed and the daemon recognises the services that it has been assigned, the daemon then decides whether it can actually perform the service.

In this example, we assume that the daemon is willing to execute the service; if the daemon is *not* willing to execute the service, either because it is too heavily loaded or the user who owns the job is not allowed to execute services on this machine, or for some other reason, the onus of selecting a new daemon to host the service(s) is on the local daemon. Although beyond the scope of this study, relocation of the service requires re-placement of the affected services to other daemons and transmitting the newly-annotated DJPL script to those daemons which have been designated to produce data that the services were to consume. Nothing needs to be sent to those daemons that will consume the data that is produced by the services on this node, as they will receive a DRAMF from the daemon that has accepted the execution request. The case where no daemon can be found to execute the affected service(s) has not been considered, nor has the case in which a number of daemons continually attempt to assign the services to each other in an infinite loop.

Assuming the daemon decides to accept the service request, an intra-daemon request is made to create the service execution process. The intra-daemon service execution process is explained in section 6.5.3. The result of the service

execution process is a number of DRAMFs which refer to the data that will be created if the results of the service are needed.

When a daemon creates DRAMFs to some future data, the service does not automatically begin executing. The agreement by a daemon to execute a service with given parameters is, however, binding. The daemon has acknowledged that if requested, it will perform the computation and return the result to any requesters. If the input data required by a service is to be created by a previous service on perhaps a remote machine, then the DRAMFs for the current service cannot be made until the input DRAMFs are received. They are needed in order to estimate the time at which the data, to which the DRAMFs point, should become available.

It is in this way that a complex processing request may be set up. This allows, essentially, resource reservations to be made for daemons that will participate in the processing of a request. Thus, while a processing request may be expressed as a network of services connected by data transfer, culminating in the production of some desired data, its execution is constructed in reverse order. Unlike other models of resource reservation, multiple services can be reserved on a single DISCWorld node simultaneously.

7. **R sends d_f to C.**

   After the DRAMFs are created by the daemon, they are forwarded to those remote nodes that may use them as inputs to subsequent services. Because the complete DJPL script is sent to all participating DISCWorld nodes, the list of recipients is easily generated from the parsed script.

   In the DISCWorld model, the partial results of processing requests are deemed to be as significant as the final results. Therefore, in addition to sending the DRAMFs to the nodes that may use them, they are also sent to the daemon or client which submitted the processing request. In the case of the user client, DRAMFs corresponding to all the partial products are returned – if the user wishes to view the contents of the DRAMF, they can perform a Copy_Data operation on the DRAM.

8. **C inspects d_f to begin execution.**

   In fact, the method by which the user client or daemon starts the computation is by performing a Copy_Data or Inspect operation on the DRAMF. As we have

seen, all DRAMs point to the remote node which has the data they refer to. The Copy_Data operation causes the data to which the DRAM refers to be returned to the local client (or daemon) as soon as it is available.

As a DRAMF represents the result of a remote computation that has not begun execution, it is started, and the result is returned when available. The way in which services are actually executed within a daemon is discussed in section 6.5.3. When the DRAMF is inspected by the client or any other daemon, a request is sent to the producing daemon, which initiates the computation.

9. **R inspects d_s, downloading the service byte-code if needed.**

   While this step and the next are implemented in parallel, for the sake of this illustrative example, they are separated into separate steps.

   When executing a computation, the first step is to make sure the service that is to be executed is available locally. Therefore, if the service is not local, the DRAMS which points to the service may need to be inspected. As previously mentioned, inspecting a DRAM causes the data to which it refers to be copied to the local DISCWorld node.

   As mentioned in section 6.3, a DRAMS has two associated arrays, detailing the service's input and output parameters. The array contains objects of type DRAM Parameter, `DRAMP`: the service's name for the parameters, a description of the parameter, and also the type of the parameter. The reason for including the type of the parameter is two-fold: firstly, it allows the type of an object that is to be used as an input or output parameter to be checked by the daemon; and secondly, when making estimations on the size that an output object may be (for transfer-time estimations), the type information is very useful.

10. **R inspects d_{d_1} and d_{d_2}, downloading the data if needed.**

    As in the previous step, the data that is to be used in a service needs to be made local. The DRAMDs representing the data is inspected, causing the data to be downloaded to the local node. In this example, R already hosts the data D_1, so it is not downloaded.

    It is at this (parallel) step where the optimisation of the processing request can be performed. If the daemon is aware of an alternate source for the data that is required, it can request that the data be downloaded from the source that will provide the data in the least amount of time. By virtue of a global naming

scheme, the daemon does not have to verify that the data referred to by two different DRAMs of the same name is equivalent. The next step in processing is not started until all the input data for the service is available.

11. **the service s is executed with parameters d_1 and d_2 and the result, f is produced.**

    When the data and service byte-code are available, the service is executed with the appropriate parameters. The service-specific name for each parameter is stored within the DRAMP object. In a Java-beans style, this name is used to locate a method in the service with the name `setX`, where `X` is the name of the parameter. The method is used to assign the value of the data to the service. The service's `run()` method is invoked, and an equivalent `getX` method is used for each output parameter.

    In this case, there is only a single output, f. When the service has completely finished creating the output object, it is added to the daemon's store and can be used in further computations.

12. **R returns the outstanding request for d_f to C.**

    The method by which the computation was initiated was by a request to inspect the DRAM corresponding to the final service output. When the output object is created, it is sent to the requester. In this case, the requester is the client C, but it may well have been another daemon wishing to use the data as input to a further service.

Optimisations are allowed (encouraged, in fact) on the services used to satisfy a processing request. The server may be aware of one or more servers that already possess the data which is to be created, or have supplied DRAMFs to the data sought. If the data is available, and the time spent transferring it will not exceed the time spent waiting until the DRAMF can be satisfied and the data returned, then the decision may be made to transfer the data from an alternative source. By the same reasoning, if there exists another DRAMF pointing to a different data source, and the estimated time is lower than that returned, the alternative DRAMF may be dereferenced in the expectation that either the computation has already been requested by another holder of the DRAMF, or the data should be available sooner due to the lower estimated arrival time.

Using partial or final data products of previous processing requests allows the possibility of pruning the execution schedule for the current processing request. Thus, if there is no need to re-compute a data product, then the system will avoid it if possible. Of course, the situation may arise where a user has inspected a DRAMF corresponding to a final data product, causing the computation to be started, and at the same time inspecting a DRAMF to a partial data product. If the DRAMF that is an input to the service which is to produce the final data product has an estimated time greater than a cheaper source of the same output data, then the data may be retrieved from the alternate source, even though by inspecting the partial product DRAMF, the user has dramatically reduced the time it would take to retrieve the data. This case is not addressed in the current implementation – we simply make a best-estimate of the execution times, and settle for that.

In the prototype version of the daemon, there is no maximum time limit between when a DRAMF is returned from a server and the execution must be started. In future implementations, depending on the cost that the user is willing to incur, multiple requests may be sent out to competing data sources, in order to achieve the fastest possible turnaround time for the processing request.

Using DRAMFs to represent the data allows the system to judge whether it is more economically feasible to wait until a result that is to be computed at one node is available than retrieve the data from another node, where it already exists but may be expensive to access. The retrieval of data from another node may have one of two effects: either the data the node was to produce now becomes redundant and a waste of processing cycles, or the data is scheduled for use by a subsequent service (after the service has found the data from another source). If the result is no longer needed, depending on the node's management policy, and usually based on the load of the node and its storage capacity, it may choose to proceed with the computation on the chance the data will be needed within the time frame that the node is willing to store results for, or may decide to cancel the production of the result. In turn, in the event of cancelling the production of the result can be propagated up through the list of servers scheduled to create the data that is no longer needed. Of course, the creation of data is only possible if there are no other servers that use the data to be created. Thus, the execution tree is pruned so that only the services that *create data* and those that create the data that is *required*, are actually executed.

Figure 43[2] shows a graphical user client through which processing requests can be

[2]Thanks to Jesudas Mathew for this screen-shot.

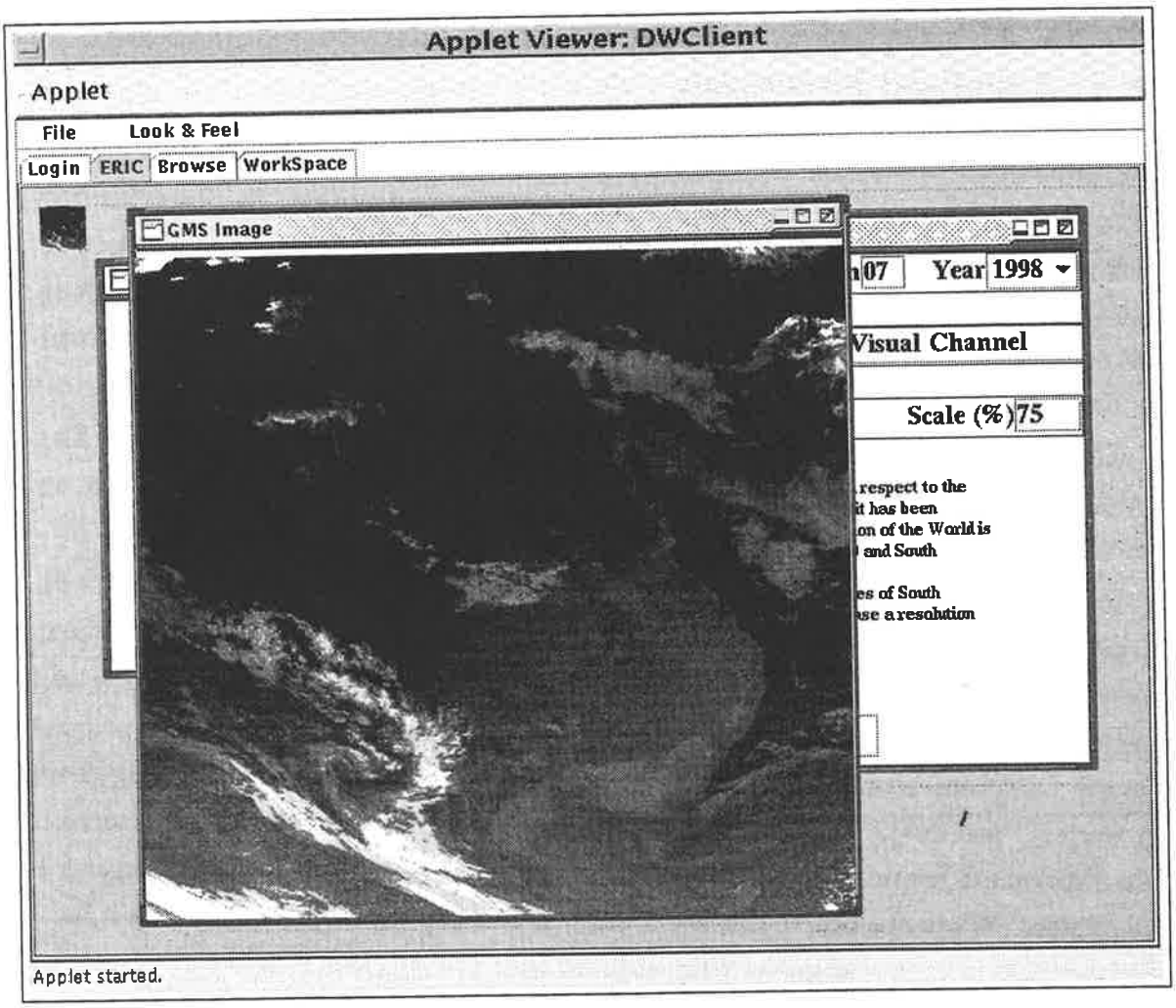


Figure 43: A graphical user interface client to DISCWorld. Services are represented by DRAMSs. They contain metadata on the services which can be used to construct complex processing requests. The user provides parameters with which services are executed. DRAMFs and DRAMDs are returned to the client, which can be inspected, returning the data they represent. The icon in the top left corner of the workspace represents a DRAMD corresponding to the large image. The occluded window on the right is the GUI to a service, again represented by a (hidden) DRAMS.

submitted to the DISCWorld prototype. Foremost in the figure is the data produced by the partially-occluded service. The icon in the upper-left corner of the workspace represents a DRAMD, which produces the main image when inspected.

6.5.3 Localised Execution

This subsection describes the way in which scheduling in the DISCWorld is implemented, from the perspective of the DISCWorld daemon. It describes the behaviour of the prototype system from a local-node viewpoint. The subsection contains a description of the DISCWorld daemon architecture, and the components that comprise the daemon; a description and purposes of the classes used to implement the daemon; and an example of the way in which the daemon operates.

The DISCWorld daemon (DWd) architecture is shown in figure 44. The components of the daemon are named according to a restaurant motif, which we feel accurately describes their functions.

In a typical restaurant, there are several types of staff, each with their own specialty. The manager runs the restaurant, overseeing the business, ensuring that management policies are enforced. The manager has the ability to specialise the business. For example, some restaurants specialise in Thai food, while others provide generic fare from around the world. A maitre'd greets customers, assigning them to waiters, and making sure the waiters provide the customers adequate service. The waiters are responsible for customer satisfaction, taking orders and dealing with complaints. When the orders are taken, they are rushed to the kitchen, where a head chef oversees a group of cooks. He ensures that the kitchen is adequately staffed to meet customer demand, hiring new staff and firing redundant personnel. In our model, the cooks are responsible for the creation of a single dish for a single customer. They are not at all specialised – they can create any dish known to the restaurant.

Our model departs from the traditional restaurant motif in the use of a quartermaster, which is an entity that has sole access to the contents of the restaurant's refrigerator. In addition, the quartermaster has the ability to ask the chef to begin construction of any necessary data products. Thus, the quartermaster is, in effect, the master of the kitchen. A useful analogy is that the restaurant serves (micro-waved) pre-prepared dishes from freezer if the dish is available, and engages the chef is not. The functions of the components are described below, together with a discussion of their use in scheduling processing requests.

manager The manager is responsible for the storage and enforcement of management policies. For example, the determining of the node's characteristics for the willingness to accept new service requests and to accept new service byte codes are management policies. Also, if the owner or administrator of a node does not wish their resource to be extensively used, or does not wish a particular service or class of services (within the hierarchical name-space) to be supported by their node, this may be expressed as a policy, which the manager will enforce. Thus, while not vital for the implementation of adaptive scheduling in the metacomputing environment, the manager does play an important rolê.

portal The portal is the point through which all messages incident to the DWd enter. Although it is beyond the scope of this work, it is intended that in future versions the portal will feature message-authentication services [99].

maitre'd The maitre'd is responsible for a group of waiters. If a query is made on the progress of a processing request, the maitre'd is responsible for the resolution of the query. Upon receiving a processing request, the maitre'd creates a waiter; when execution has finished, the waiter is destroyed. The maitre'd is important for scheduling, as it provides the creation facilities for waiters.

waiter The waiter is the entity responsible for a single processing request. The processing request must be parsed, and the services it contains assigned to daemons. The waiter initiates these tasks, and arranges for the distribution of annotated processing requests. Furthermore, if a service is assigned to execute on the current node, the waiter is responsible for initiating the execution of the service, and the propagation of service output futures. The waiter is vital for scheduling, as it provides the mechanisms by which the processing request (expressed in DJPL) is parsed, annotated, distributed, and local execution is initiated.

store The store is an idealised database, which contains all the DWd's current knowledge. Information on the data, services, remote nodes and interconnection networks are stored here. Both DRAMs to data, and the data themselves are stored. The manner in which the store is implemented is not specified – some DWd's may use a relational object-oriented database, while others may use a minimal array-based or Vector-based solution. The store is very important for the adaptive nature of scheduling in the DISCWorld model, as it allows the partial results of previous requests to be stored and re-used.

quartermaster The quartermaster is the interface to the store. It can fill outstanding requests for data (or service) items. The quartermaster is the DWd component that makes remote daemon requests to retrieve information. The quartermaster is vital to the implementation of scheduling in DISCWorld, as it allows blocking requests for data to be made, and allows the proper updating of system state information. In addition the quartermaster is responsible for interaction with other daemons to perform resource discovery. DRAMs are exchanged by daemons in order to update system state information.

chef In the DISCWorld model, the chef represents the head of the kitchen. The chef is in charge of the creation of cooks to execute services, and the termination of cooks once they have performed their duties. The chef controls the activation of cooks, and ensures that there are no more cooks running than can be adequately handled by the node on which the daemon is running. The chef is an important component in scheduling, as it provides a measure of load-limiting for cooks. The chef monitors the effective load on the local daemon. Cooks are only able to begin executing services whilst the load is below a threshold represented by a management policy.

cook The cook is the most fundamental component of the DISCWorld architecture. This is the component that actually executes the services to fill a processing request. It is vital for scheduling because it provides the framework in which services are executed. Cooks are able to execute any service in the DISCWorld providing the daemon possesses a DRAMS to the service (and the code is movable or already local).

For each of the DWd components in the DISCWorld model architecture, the corresponding Java classes in the implementation are described. The classes are shown in figure 45. Encapsulated boxes refer to Java inner classes, an abstraction which allows the inside class to inherit the class- and instance-variables and methods from the outer class. Inner classes also serve to restrict the visibility of classes from each other.

The executable class file is `daemon`. In the current implementation, it assumes the most basic function of the portal, that of message receipt. No authentication is attempted in the current version. All messages sent across the network are encapsulated in a `GlobalThaum`, which is an object wrapper that provides source and destination addresses for DWd's, and whether the message is a request or

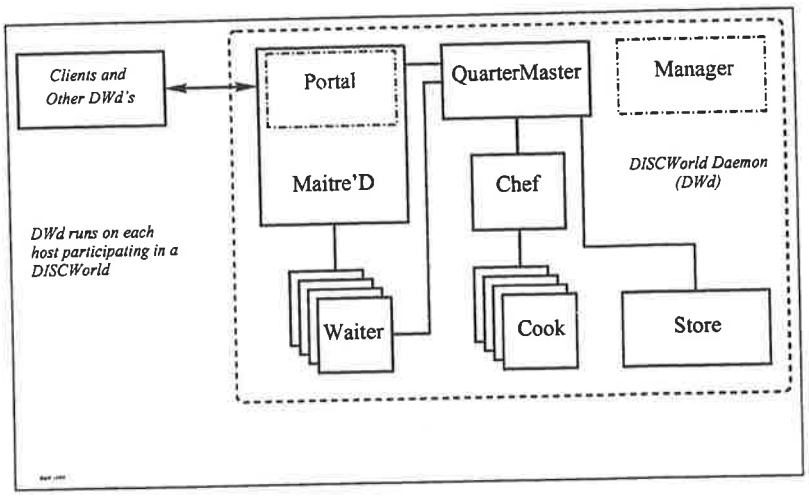


Figure 44: Components of the DISCWorld architecture that are necessary and important for scheduling and placement of services and data. While the portal and manager are necessary for a complete implementation of the DISCWorld model, they are not implemented in our prototype.

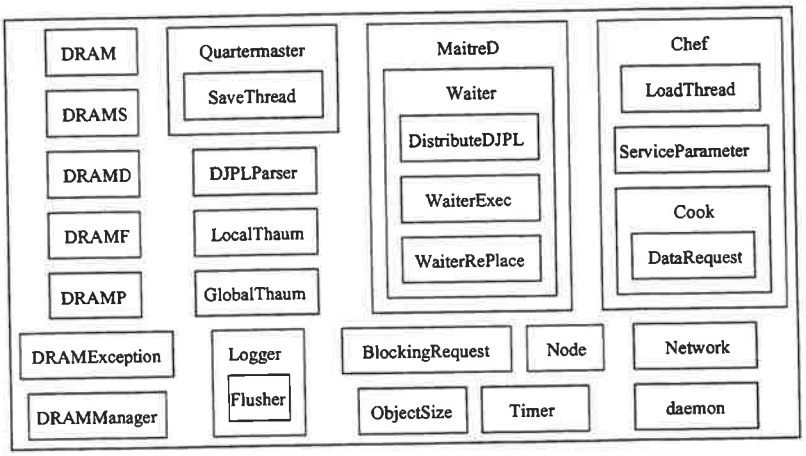


Figure 45: Java classes and inner classes which comprise the prototype DISCWorld implementation. Inner classes are drawn inside the class which encapsulates them.

not. Intra-daemon messages use a light-weight version of the GlobalThaum, the `LocalThaum`, to encapsulate messages. The LocalThaum contains the name and type of the encapsulated object. To facilitate debugging and analysis of the DWd, a separate class, `Logger`, is used. This has a separate thread that periodically flushes accumulated messages to a local file, thus lessening the impact on the remainder of the daemon.

It can be seen in figure 44 that two objects communicate with the DWd's portal: the `maitre'd` and the `quartermaster`. The maitre'd is implemented as a class which contains the `waiter` inner class. The waiter class contains three inner classes: `DistributeDJPL`, `WaiterExec`, and `WaiterRePlace`. DistributeDJPL distributes annotated DJPL scripts to all nodes which are assigned services in the current processing request. WaiterExec executes services assigned to the local daemon, and WaiterRePlace forces a complete re-annotation of the DJPL script in case a more economical solution can be found.

The quartermaster class only has one inner class, `saveThread`, which performs the periodic dump of the store to disk. In addition, the quartermaster is nominally responsible for the monitoring of `BlockingRequest` objects. Within the quartermaster, the `chef` object is initialised, which has the ability to organise and begin the execution of services. The chef has three inner classes, `cook`, `LoadThread`, and `ServiceParameter`. ServiceParameter is an object which is used internally to match parameter names to values. The cook performs marshaling of parameters and the execution of services. Furthermore, the cook has an inner class, `DataRequest`, which is used to perform a blocking request for data, encapsulated within a thread. This encapsulation is used to allow a thread join to be performed – ensuring that all parameters have been correctly marshaled. The chef also creates a `LoadThread` thread, which measures elementary local-node load information through the `Timer` object.

All data and services within the DISCWorld are encapsulated within a `DRAM` object. The DRAM object is refined into a service DRAM, `DRAMS`, a data DRAM, `DRAMD`, and a future DRAM, `DRAMF`. The DRAMS has an associated object to store parameter information, the `DRAMP`. `DRAMManager` is an interface to which the daemon adheres. It ensures that the methods for DRAM inspection and movement are available on the local node. `DRAMException` is used to signal error conditions. The `ObjectSize` class is used to measure the size of an instance of an object, so that the network transfer time can be made more accurate.

The remainder of this subsection describes the operation of the DISCWorld daemon. Figure 46 shows the classes from figure 45 that are created as threads. Multiple instances of a thread are indicated by the use of n in the diagram.

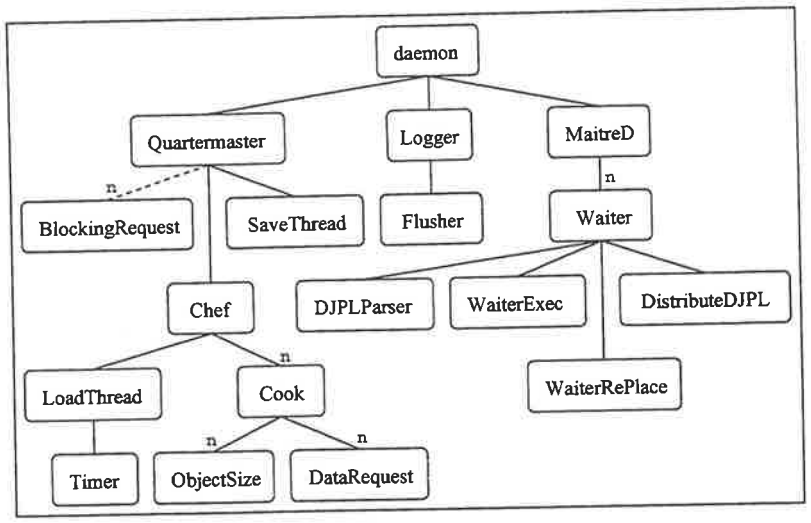


Figure 46: Java threads and their hierarchy in the prototype DISCWorld implementation. There is a single instance of each thread executing in the daemon, with the exception of those marked with n, which are created on demand and terminated after use.

All messages in the DISCWorld are encapsulated within a GlobalThaum. When a GlobalThaum arrives at a DWd, it is received by the daemon class. The contents of the GlobalThaum are checked. If the message is a request, then a BlockingRequest is created to the quartermaster. If a message is not a request, it may nominally be called a response, although this terminology is not completely appropriate, as it does not convey the case where the information is unsolicited. The LocalThaum, encapsulated within the GlobalThaum, is routed to either the quartermaster or the maitre'd, depending whether the contents of the response message are a DRAM or DJPL script, respectively. If the message is not encapsulated within a GlobalThaum object, it is dropped and a log of the message is kept.

When a DJPL script is sent to the maitre'd, the maitre'd creates a new waiter to process the request. In the DISCWorld architecture model, the maitre'd allows clients to query the execution status of their processing request; this is not implemented in the current prototype.

The waiter creates a DJPLParser object, which parses the DJPL script, determining if it has been completely annotated. If not, a schedule is created in accordance with the algorithm presented in chapter 5. If the DJPL script is further annotated, DRAMs corresponding to the services and data are sent, with a copy of the annotated script, to the chosen nodes. This is performed by the DistributeDJPL object, which ensures that all the necessary DRAMs are distributed. The waiter then creates a WaiterExec object, which traverses the annotated DJPL script, searching for services that have been assigned to the local DWd.

If the local DWd has been assigned to execute any services as part of the processing request, the WaiterExec object makes a request to the chef for it to be executed. When the chef accepts the execution request, DRAMFs, corresponding to the outputs of the services, are created. The DRAMFs are sent to the nodes that have been assigned to use them (as per the annotated DJPL script), and also to the client that submitted the processing request.

When a DRAM is sent to the quartermaster, the store is checked to see if the object is already in the store. Addressing of objects within the store is via their name and the host from which they came. If the data is currently in the store, the newly-received DRAM is treated as an update, and the store's object reference is updated. Otherwise, the DRAM is added to the store. In the case where a daemon currently has a DRAMF to a future result, and the daemon receives a DRAMD to the actual result, the DRAMF is removed and the DRAMD inserted into the store. The addition of new data to the store, or the updating of current data may cause a BlockingRequest to be satisfied. The only other objects that are contained in the quartermaster's store are Node and Network objects. These are used to represent the state of the global system, as discussed in chapter 5.

If the BlockingRequest is for data that is to be created at the local node (i.e. the subject of the BlockingRequest is a DRAMF), then the quartermaster sends a message to the chef to begin the computation of the requested data product. When the chef is first created, there are no cook threads. When a request for processing is received, the chef first checks that no currently-active cooks are performing the requested service (with the same parameters). If not, a cook is created.

The processing request, from which this service was initiated, contains the name of the node at which the input data was created. The cook then queries the store, through the quartermaster, to ensure that all the necessary data, from the specified sources, is present on the local node. The data can exist as a DRAM with no local

object (i.e. the DRAM has been received, but has not had a Copy_Data operation performed on it yet).

It is at this point in the execution that optimisation can be performed. A cook attempts to achieve service execution in the quickest amount of time. If the data that is to be used as input is available from another node, or is already present on the local node, or the current node has a DRAMF for the same data, with a sooner estimate of availability, the cook may decide to use the alternate data source. One of the assumptions of the DISCWorld model is that the size of a data item is the same no matter where it is created. Thus, if a daemon has a DRAMD *and* a DRAMF to the same data, the estimated size of the DRAMF can be updated to the (exact) size of the DRAMD. If the data is available from multiple sources, the following minimisation is performed:

$$t_{\text{DRAMF est wait}} + network_{\text{lpt}} + data_{size}/network_{\text{bpt}} < network_{\text{lat}} + data_{size}/network_{\text{bat}}$$

where *lpt* is the latency as measured from the promised source to the local (target) node, *bpt* is the bandwidth between the promised source to the local node, *lat* is the latency as measured from the alternate node to the target node, and *bat* is the bandwidth from the alternate node to the local node.

From the DRAMDs corresponding to the input data, an estimate is able to be made on the time that it might take to marshal the parameters and execute the service. DRAMFs corresponding to each service output are added to the store and returned to the requesting objects. After the DRAMFs have been returned, the cook then registers itself with the chef and enters a wait state. The chef is then able to re-awaken the cook if any of the DRAMFs are inspected. This starts the cook marshaling parameters, which is done by inspecting each input DRAM. After the DRAMs have been inspected, and their local objects returned, the cook then advises the chef that it is now able to actually execute the service, and again waits.

When a service is requested to be executed, we use the real-life analogy of someone asking for a quote to perform a job. By accepting the request, and supplying a value, the daemon has guaranteed it will perform the service if required. Execution is only begun when one of the results of the service is requested. The execution of services is split into a number of stages to allow resource utilizations to be properly managed. The first phase of execution, parameter marshaling, is assumed not to be expensive, in terms of resource utilizations and computation, because it is simply initiating data

transfer as a result of DRAM inspection. As such, there should be little impact on the local system if there are many requests executing in this phase. The second phase, the actual execution of the service, is expected to be computationally expensive. For this reason, the daemon controls the number of requests executing in this phase.

The timer thread produces a value which represents the current number of *spare cycles* on the machine compared to the total number of spare cycles that have been measured. Thus, if the machine is very lightly loaded, the number of spare cycles will be high; if the machine is experiencing high load, the spare cycles will be low. The ratio of current spare cycles to the maximum number measured gives an estimate of the load.

When the load measure is beneath a certain threshold, the chef chooses a cook to begin executing. In the current implementation, the queue for execution is modeled as a Java vector. Jobs are always taken from the front of the vector. Thus, while there is no *guaranteed* ordering of the execution of services, it seems to work in practice. In the current implementation, there is no measure of priority – all services are equally as important, and we do not use any hard- or soft-real-time deadlines. Once the service has finished executing, the output DRAMDs are added to the store. This, in turn, may satisfy a waiting BlockingRequest or another service waiting on parameter marshaling.

In the current implementation, the quartermaster assumes some of the duties of the manager in the DISCWorld architecture model. Policies such as the period between copying the store, period and frequency of testing for system load, and the threshold for creating new jobs that will be controlled by the manager, are currently implemented by the quartermaster.

6.6 Summary

In this chapter, we have discussed the implementation of scheduling in the DISCWorld prototype. In describing the implementation, we have discussed two different types of request execution models, data push and data pull, and how they are applicable in the model under consideration. An important feature of the model is the existence of a global name space, with which objects may be named in an platform-independent manner. We have introduced the method by which global names in the DISCWorld system are generated and are used. While the system that has been implemented and

used for experimentation may prove to be ultimately infeasible in a large-scale system, its use is sufficiently generalised that it may be replaced with little inconvenience.

We have introduced the DISCWorld Remote Access Mechanism (DRAM), which is a rich pointer to data and services residing on DISCWorld nodes. DRAMs are a higher-level construct than memory pointers (or references) found in many programming languages; DRAMs can refer to bulk data items and service byte-codes, which may be traded between machines and the data that they point to retrieved. DRAMs have the interesting property that if the data to which they point is unavailable (e.g., because it has been garbage-collected or the machine from which the DRAM originated has failed), the data can be reconstructed, albeit at far increased cost. Thus, while a DRAM points to remote data, it also contains the *recipe* with which the data can be reconstructed.

In order to facilitate the implementation of large-scale distributed service execution, we have extended the DRAM concept to allow data that has not yet been created to be manipulated. Future data is encapsulated in the DRAM Future (DRAMF). DRAMFs are used extensively to optimise service execution time and cost, especially as there are no guarantees on how accurate a node's global system state information is. Optimisation is achieved through data and service re-use on a per-node basis.

The software components that make up the DISCWorld daemon, and their purposes, are described. An analogy is presented that allows the easy and intuitive understanding of the daemon components. The general behaviour of the distributed environment is described from the levels of both autonomous DISCWorld daemons and of intra-daemon communications. To aid in the description and understanding of the behaviour of the system, a simple processing example is discussed.

The DRAM abstraction described in this chapter is similar to a number of existing technologies, such as Nexus' global pointers and remote service requests [73], and CORBA's Internet Inter-Orb Protocol(IIOP) and Interoperable Object Reference (IOR) [176, 177]. With the exception of CORBA IOR, the fundamental difference between DRAMs and the other technologies is that they are targeted at fine-grained object references, whereas DRAMs are more coarse-grained. In addition, DRAMs are long-lived objects, unlike Nexus' global pointers and CORBA IORs. When a server that stores an object is restarted, if the object is recreated, it is unlikely that the object will be created at the same location in memory. Thus, although the object to which the reference points is present, any remote pointer that addresses the memory

at which the object was stored will no longer be valid. DRAMs avoid this problem by naming the data (or byte code) that is stored on a server in a memory location-independent manner. Thus, it is the server's responsibility to ascertain whether there is a match between the named data and an actual object.

The IOR is CORBA's closest mechanism to DRAMs. IORs provide a mechanism whereby an object reference can be sent across Object Request Broker (ORB) boundaries. They provide a persistent reference to a server object, which can be transferred around the distributed system. When an IOR is dereferenced, it transparently creates an instance of the object to which it points on the object's local ORB. All further interaction is with the instantiated remote object. In contrast, DRAMs contain enough metadata to reason about the referenced object, and when the DRAM is dereferenced, the object is downloaded to the local host.

In the next chapter, we present the performance analysis of the prototype DISCWorld system. We also present a discussion of the limitations of the current implementation and discuss suggestions for future implementations.

Chapter 7

DISCWorld Performance Analysis

The services that are presented for performance measurement perform the same functions as those found in the ERIC prototype [129] (see appendix B). As such, a direct comparison may be made between the performance of the services using the DISCWorld daemon and ERIC. Both ERIC and the implemented services provide remote access to a repository of GMS satellite [133] imagery. In addition to providing browsing access, they allow processing operations to be invoked on the stored images.

The main difference between the two approaches is that the ERIC program involves processing on the server-side only. The program is implemented as a Perl common gateway interface (cgi-bin) [100] script. The script interfaces to user-level command-line programs which communicate by the use of shared file systems and pipes. No distribution of the command-line programs is possible unless it is explicitly incorporated by the author of the script. The user composes a query using a HTML forms interface, and the result is returned as another HTML page. Although ERIC implements basic caching, only resultant images, metadata and MPEGs [134] are stored. No partial results are stored, and final results cannot be used as inputs for further processing. Other limitations of the ERIC prototype are discussed in appendix B.

In contrast, the DISCWorld daemon allows processing to be performed at both the client-side and server-side. For example, the scheduling placement algorithm may decide that simple imagery operations can be more efficiently performed at the client-side than the server-side. Services can be written as pure Java or may be implemented as a wrapper around a native program (using either JNI [148] or system calls). Objects can be sent between nodes. Consequently, there is no need for common file-systems. The DISCWorld daemon stores *all* results of services, whether they

are the final results of the user's processing request or the results of intermediate computations. Intermediate results can be used in subsequent processing requests since the DISCWorld model assigns canonical names to all objects..

7.1 Example Services

For performance analysis, we focus on three example services: `findGMSImage`, `processGMSImage`, and `createMPEG`. FindGMSImage is very similar to one of the fundamental operations found in ERIC: it invokes a shell-script to retrieve an image from the spatial imagery archive. After finding the image, it is loaded into the DISCWorld daemon using services built from the methods in the Java Advanced Imaging (JAI) [217] API. The image is then accessible via its DRAM. The input parameters to the service are the date and time at which the image was created, and its spectral channel. This service invokes a native program, and it is marked as "not movable" (as described in chapter 6); it cannot be transferred between nodes.

ProcessGMSImage uses services from the JAI API to crop and scale the GMS Image. It performs the same imagery operations as found in the ERIC. The inputs to the service are the GMS image, the area to be cropped, and the zoom scale of the resultant image. As this service is written completely in Java, it can be transferred between nodes.

CreateMPEG is a service that creates MPEG animations from sequences of images. This service invokes the `mpeg_encode` [87] program as a system call. Due to the way in which the program is implemented, it is necessary to write the images and an associated parameter file to a temporary disk area. The program also writes the newly-created MPEG to the disk, where it is ingested into the system using services from the JAI API. Like the findGMSImage service, the createMPEG service is deemed "not movable".

Three computers were used for testing the performance of the DISCWorld daemon: a dual-processor Sun Enterprise 250, (called `lerwick`) with 256Mb of physical and 300Mb of virtual memory that runs Solaris 2.6 and Java 1.2.1; a dual-processor Celeron, (called `banff`) with 256Mb of physical and 1Gb of virtual memory that runs Solaris 2.6 and Java 1.2.1; and a single-processor Pentium II, (called `geronimo`) with 64Mb of physical and 64Mb of virtual memory that runs Windows NT 4.0 and Java 1.2. Lerwick and banff are connected via a 100Mbps network; geronimo is connected via a 10Mbps network.

Execution Environment	Time to complete each service (ms)		
	findGMSImage	processGMSImage	createMPEG
Unix command line	4000 ± 490	n/a	5720 ± 40
Java on local host	7011 ± 1100	7597 ± 1300	12473 ± 120
RMI, local client	9858 ± 5600	10869 ± 6000	14545 ± 2300
RMI, remote client	10922 ± 2700	11114 ± 2900	15916 ± 2500
DISCWorld, local client	24637 ± 13000	12820 ± 100	27804 ± 3400
DISCWorld, remote client	25322 ± 12000	12950 ± 1000	29657 ± 4700
local DISCWorld cache	1272 ± 46	1612 ± 300	1700 ± 300
remote DISCWorld cache	8114 ± 200	11399 ± 200	11742 ± 200
ERIC, not in cache	n/a	1700 ± 100	48000
ERIC, in cache	n/a	26000 ± 1000	26000 ± 1000

Table 8: Comparison of overheads when individual services are executed by different environments. The execution time of the services implemented as command-line programs are measured so the overheads of the Java system can be measured. The time taken for a pre-computed result to be retrieved from the local daemon, and from a remote daemon via a slow 10Mbps link are also shown. Measurements of the ERIC system, from [127] are shown for comparison. All times are based on at least 10 measurements. Variances are based on a least-squares linear fit of the measured values. The error in the timers used to collect timing information is that of Java's `currentTimeMillis` method, which does not have an accuracy of exactly one millisecond [157].

Table 8 shows the time that each of the services takes to run on lerwick under a number of different execution environments. The Unix command line measurements are the execution time of the command-line programs. ProcessGMSImage cannot be measured in this way as it is written purely in Java; it cannot be directly executed without the creation of a JVM. All services within DISCWorld are encapsulated by a service wrapper, which provides JavaBean interfaces to the methods of the service. A service's `run()` method is invoked to execute the service. The run-time penalty due to Java can be calculated by invoking the service from a thin client on a local machine. The run-time overhead on findGMSImage is approximately 3000ms. We attribute this to: the necessity of creating an instance of the service in the JVM; invoking the service's `set` method for each parameter; and forking a new process to execute the shell-script. This overhead is increased with the createMPEG service because of the need to write each of the input images to disk before executing the mpeg_encode program.

It is useful to compare the performance of each of our example services when

executed as stand-alone servers in an RMI environment. As table 8 shows, the additional overhead of each service when invoked via RMI is approximately 2000ms. The increase in execution time can be attributed to service brokering by the RMI registry and the fact that each of the services is executed in their own JVM. This overhead is not substantially increased when the RMI client is non-local.

The times quoted for the DISCWorld daemon are the total elapsed time from when the user client submits a processing request to when the client receives the final DRAMD. This includes the time taken to: transmit the processing request to the daemon; parse and place the service; return the DRAMF to the client; for the client to inspect the DRAMF; for the service to be executed; and for the DRAMD to be returned to the client. As the majority of the time spent executing a user processing request is at the server-side, the additional time taken to submit a request from a remote client is relatively small. Clearly, the DISCWorld daemon introduces some overheads, which when amortised over large services, become relatively inconsequential.

As previously mentioned, all results and partial results in DISCWorld are cached. This proves to be useful when a client requests an object that has already been cached. As illustrated in table 8, requesting a cached object from a DISCWorld daemon's store is extremely efficient. In contrast to the remote clients using RMI and DISCWorld, which were measured using a fast (100Mbps) network, the results of the remote retrieval from the DISCWorld cache are across a slow (10Mbps) network. The retrieval times are dominated by the network characteristics. The time is comparable to the time taken to execute the service using a thin Java client on the faster machine, or using RMI on the faster machine. Use of the remote client to request information from the local DISCWorld daemon illustrates the re-use of DRAMDs and DRAMFs inside the daemon – in this example, the original processing request that caused the data to be created was made from a client on the node local to the daemon. This further emphasises the usefulness of the DRAM mechanism, whereby the data may be further processed before being returned to the client, which may be connected via a very slow network.

In addition to the timing information presented in table 8, further measurements of the ERIC system are presented in [127]. While timing information for the findGMSImage alone service is not available for ERIC, the times for ERIC's processGMSImage service include that taken by findGMSImage. No standard deviation is provided in [127] for the time to create an MPEG at the resolution

and zoom that we consider in this analysis. The figure given for the time taken to retrieve a cached MPEG is that taken to retrieve any object from ERIC's store.

The performance of the DISCWorld daemon is very dependent on the background load of the processors. The performance of Java is very susceptible to these effects because not only do the Java application's threads need to be scheduled by the JVM, but the JVM needs to be scheduled by the host operating system.

Table 9 shows the performance of the DISCWorld daemon in the case where a single server is chosen to be used by the placement algorithm. The first two columns of this table show the effects of adding a second service which is independent of the first. The mean time to return DRAMFs, corresponding to the outputs of the services, scales linearly with the number of independent services. However, there is a high standard deviation in the DRAMF creation time for the case in which there are two independent services. The increase in DRAMF creation time may be attributed to additional parsing necessary for the extra service, and also to increased contention for the quartermaster's synchronised store object, as described in chapter 6. Interestingly, while the mean time shown to execute two services is less than that for a single service, the standard deviation is larger. Within the bounds of experimental measurement, these values appear to be the same. This is feasible because the daemon is multi-threaded, and it able to execute many services simultaneously.

	Total elapsed time, as observed by client (ms)		
Action	Single service	Two independent services	Two dependent and two independent services
DRAMF creation	380 ± 90	754 ± 560	1608 ± 110
Service execution	15548 ± 3400	14522 ± 5300	28879 ± 8800
Cache retrieval	1169 ± 60	1217 ± 70	1429 ± 100

Table 9: Performance of DISCWorld prototype using multiple processing requests. In the case of two services, two independent findGMSImage services are requested; in the case of four services, two independent instances of a findGMSImage and processGMSImage service are requested. The DISCWorld daemon processes all independent services concurrently.

The last column of table 9 presents the results of a client submitting a processing request in which there are two independent requests, comprised of a processGMSImage service that uses the output of a findGMSImage service. This may be compared with the second column of the table, in which two independent

findGMSImage services are requested. The time taken to create and return DRAMFs is found to scale linearly with the total number of services requested[1]. The addition of the two processGMSImage services, dependent on the output of their respective findGMSImage services, causes the total execution time of the processing request to double. Thus for these services, at least in this situation, the performance scales linearly.

The situation in which more than one client simultaneously makes a processing request to the same DISCWorld daemon is shown in table 10. The time taken for the daemon to create and return a DRAMF, and to return previously-created DRAMDs from the cache, increase by approximately 750ms per additional client. Each additional client increases the time taken to execute a single findGMSImage service by approximately 7200ms. This again emphasises that the daemon scales linearly, subject to the limitations as described above.

Action	Total elapsed time, as observed by client (ms)		
	Single client	Two clients	Three clients
DRAMF creation	547 ± 456	901 ± 438	1330 ± 726
Service execution	13563 ± 1651	21678 ± 2912	28340 ± 11934
Cache retrieval	2151 ± 24	2877 ± 729	3598 ± 1209

Table 10: Performance of DISCWorld prototype using single service when processing requests are made simultaneously. Within the bounds of control, each client submits a processing request, and then inspects the resulting DRAMFs at the same time.

Multiple servers will only be used by the scheduling placement algorithm when it will result in a lower processing cost, or if the required services are only available on different servers. Of course, when using multiple servers the effects of the interconnection network become significant, as illustrated above.

As described in section 6.5.1, whenever a daemon receives a DJPL script, the script is parsed and tested to see if any extra placement information needs to be generated. If two nodes are used for processing, the first node (local to the client from which it was submitted) generates an annotated DJPL script for the request. The annotated DJPL script is then distributed to all participating nodes. When the script is parsed and the services created, no DRAMFs are produced until all the

[1]Because of the way in which the store uses memory, the maximum number of independent services able to be tested at once was seven. Each independent service was comprised of two dependent services, bringing the total to fourteen. This limitation will be addressed in future work, as discussed in section 7.3

DRAMFs that will be used as inputs to that service are received. This adds extra overhead that is not been taken into consideration when creating the schedule. This is a second-order effect, and has not been incorporated into the model.

7.2 A Detailed Example

Further to the example presented and discussed in section 6.5.2, we present a detailed example of event timing within the execution of a client processing request. We use this example to discuss the strengths and weaknesses of the DISCWorld architecture model.

Consider a processing request in which a number of GMS images are to be retrieved and processed on `lerwick`, and then transferred to `banff`, where they are used to make an MPEG. This sequence of events is shown pictorially in figure 47. An individual GMS image may be identified by the time and date at which it was made, and the spectral channel it represents [129]. A processed GMS image has the additional information of the bounding box to define the area of interest, and the zoom scale of the resulting image. The figure shows the services involved in the creation of such an MPEG; the images represent the data being manipulated at each step (although at greatly reduced resolution). When the processing request is parsed by the DISCWorld daemon, DRAMFs corresponding to the future data product are returned to the user. When the DRAMFs are inspected, the computation is started. Therefore, the arrows in figure 47 represent both DRAMFs *before* the computation is started, and DRAMDs *after* the partial products have been made. For all the image cases tested, the performance of the daemon scales approximately linearly. The case in which there are four images to be used and the processing request is submitted from `lerwick`, the following sequence of events, at the following times are observed (we ignore the following steps discussed in section 6.5.2: resource discovery, user logging onto the client, and composing the processing request):

On lerwick,

1. The waiter is created and DJPL parsing is started,
2. DJPL parsing takes 1455ms,
3. WaiterExec and DistributeDJPL threads start,
4. As all the findGMSImage services will be run on `lerwick`, and all the inputs to findGMSImage are supplied by the client, the DRAMFs are immediately created

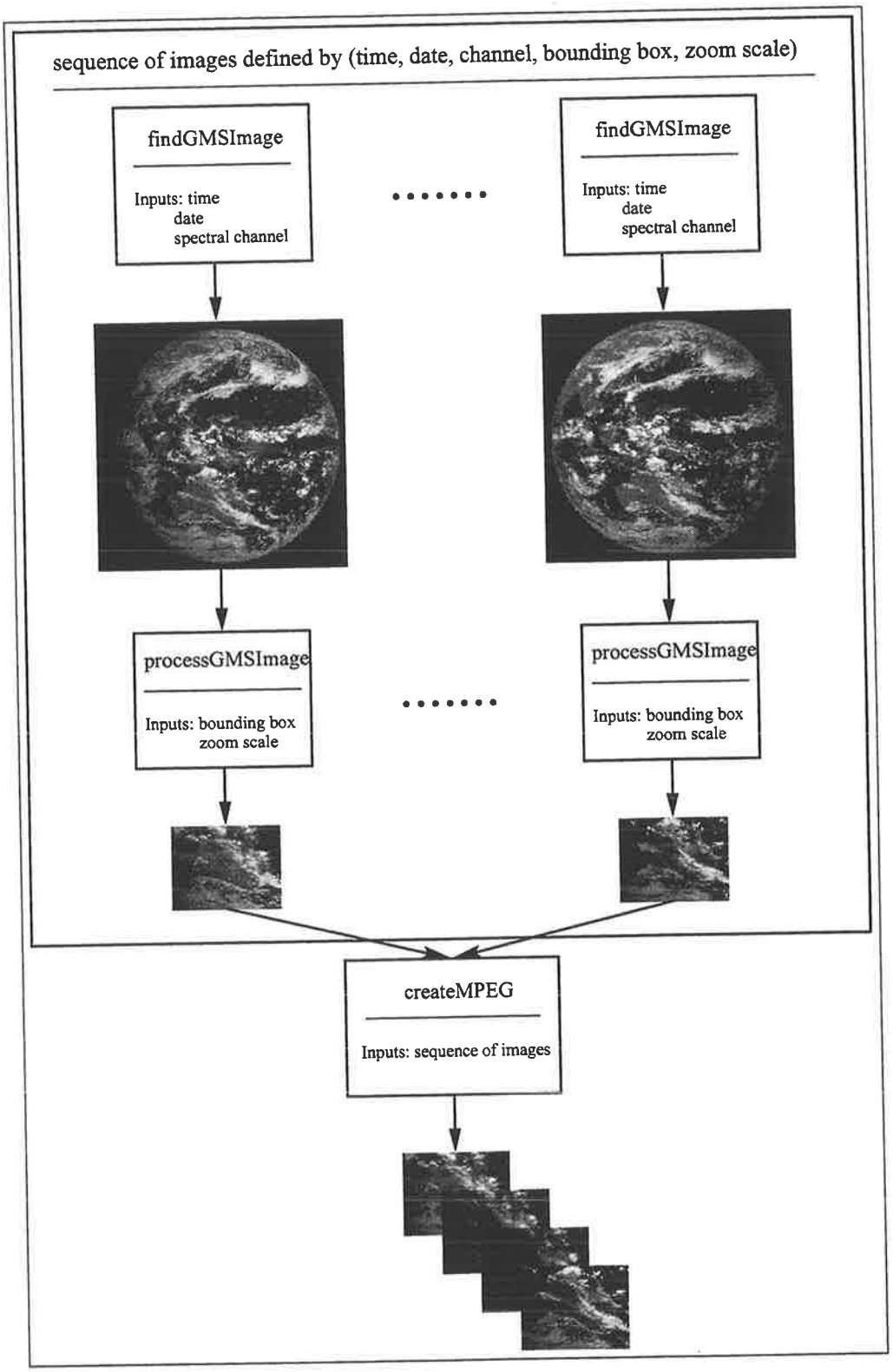


Figure 47: Pictorial representation of a complex DJPL request. A processed GMS image is defined by the tuple (time, date, spectral channel, bounding box, zoom scale). A sequence of processed GMS images is used as input by the createMPEG service. The query is constructed using DRAMDs and DRAMFs, denoted conceptually by arrows. The images show the data that is produced and consumed at each processing step.

and distributed to the servers that will use them. As the processGMSImage service will use the outputs of findGMSImage, and is on the same node, the DRAMFs are only transferred to the user client. The processGMSImage services are created; their output DRAMFs are transferred to the nodes that will use them (`banff`) and the user client. This process takes 289ms. It takes 219ms for all the servers to be sent the annotated DJPL script.

5. In total, the preparatory service creation, DJPL and DRAM distribution takes approximately 1750ms.

When the DJPL script and DRAMFs are received by banff,

1. The waiter is created and DJPL parsing is started,
2. DJPL parsing takes 6511ms,
3. WaiterExec and DistributeDJPL threads start (the DJPL script is not annotated further by this DWd, so it is not distributed to any other nodes),
4. It takes 215ms to start the createMPEG service, create the output DRAMF and send it to the client on `lerwick`,
5. In total, the preparatory service creation, DJPL and DRAM distribution takes approximately 6750ms.

The user client, running on `lerwick`, receives the DRAMF for the output of the createMPEG service, and inspects it as soon as it is received.

Banff receives the request for the DRAMF to be created

1. the service to create the createMPEG output begins to marshal parameters. Requests are sent to the data's originating server (`lerwick`).

Lerwick receives the request for the data to which the DRAMFs correspond,

1. The time taken for the findGMSImage service to execute takes approximately 28000ms,
2. The time taken for the processGMSImage service to execute takes approximately 13000ms,
3. Blocking requests for data in the quartermaster's store are periodically tested in case the thread is not awakened by a resume signal.

4. In total, the execution of the findGMSImage and processGMSImage services take approximately 127000ms. The data is returned to the requesting node (`banff`).

On `banff`, the DRAMDs corresponding to the output of the processGMSImage services are received,

1. The blocking requests corresponding to the DRAMF inspection takes approximately 145000ms,

2. The createMPEG service takes approximately 30000ms to execute,

3. The client's blocking request for the result of the createMPEG service is filled

The event sequence of this processing request is shown in figure 48. It can be seen that the majority of the total execution time is spent when the DRAMFs that are produced by the processGMSImage services are inspected. The request, instigated on banff, causes the services on lerwick to be started, and the final results returned. This example shows how DRAMs can be used to set up, and execute complex processing requests that span multiple nodes in a distributed system.

It can be seen in this example that the parsing of the annotated DJPL, as distributed by the DISCWorld daemon on lerwick, takes an abnormally large amount of time. While we are unable to pinpoint the exact cause of this anomaly, we suspect that its cause may be due to the way in which the annotations are implemented by our parsing routines.

This example shows the case in which a server has been specialised to create MPEG animations. Another example of a server specialising in a specific service is the `dploader` tool (described in chapter 4), which offers a parameterised simulation service that uses a network of workstations that are unable to run their own JVMs.

It can be seen from this example the benefits of caching partial results of services. For example, if a number of the same images are to be re-used, the total service execution time will be approximately 145000ms shorter. We have not addressed the issue of flushing cache contents; this is planned in future work. If a DRAMD with the same name is available from a different server at a lower cost, it may be retrieved from the alternate server, thus optimising the execution of a processing request.

As has been previously discussed, one of the limitations of the current implementation of the DISCWorld daemon is its poor memory management. If the cache becomes too full, the daemon will fail, losing its current state information.

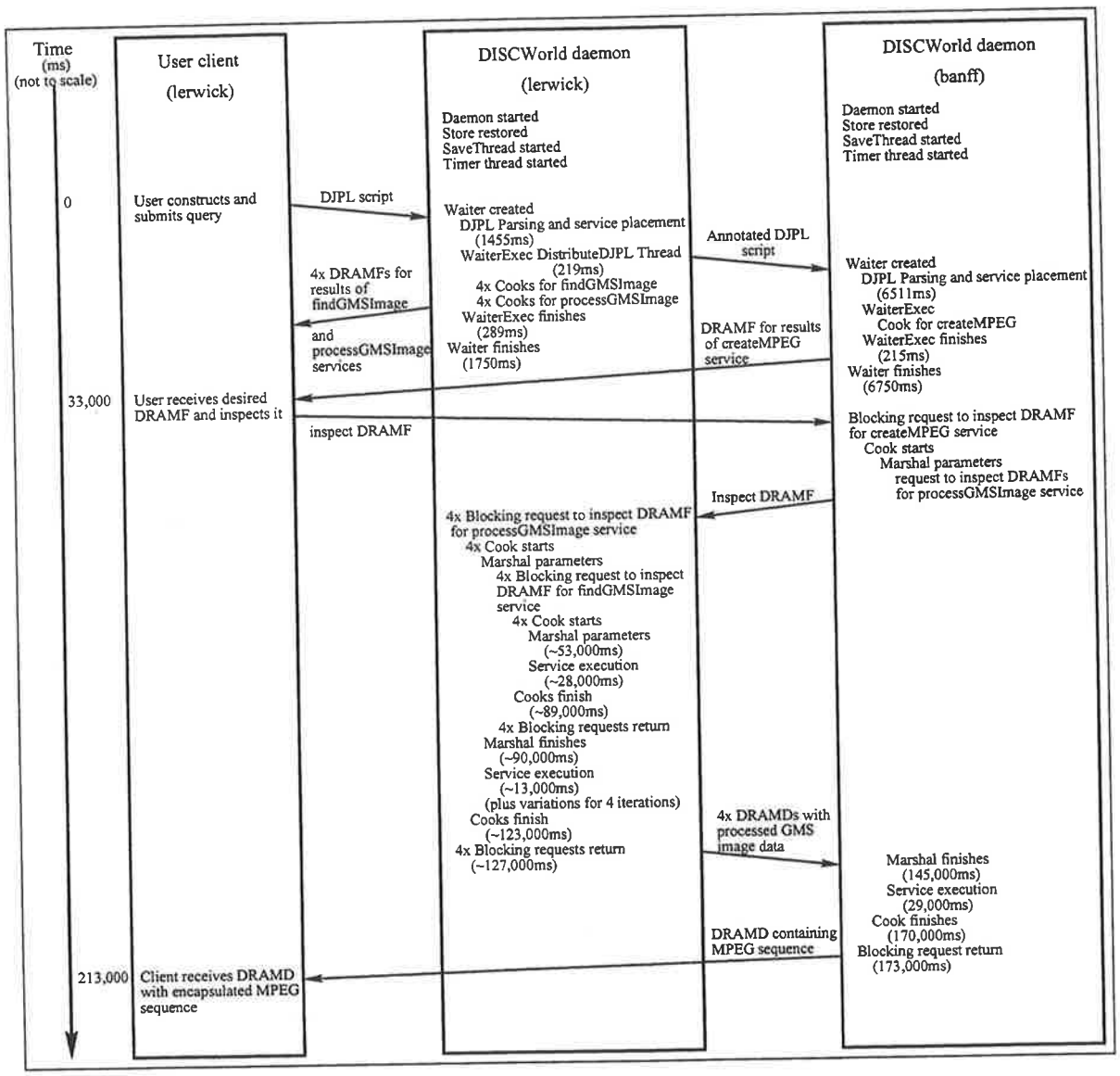


Figure 48: Event timing diagram of the complex processing request, shown in figure 47, using four images. By far, the greatest amount of time is spent in the findGMSImage and processGMSImage services. This is not unreasonable due to the penalty of co-locating services on the same node.

7.3 Performance Considerations

The current DISCWorld daemon implementation relies heavily on information about available nodes and interconnection networks. In the current implementation, if both the node and interconnection network information are not available, a node is not used. The reason for this is that while a node may be available all of the time and may have a very short waiting time, its interconnections to the remainder of the network may introduce a processing bottleneck. Future versions are intended to perform an on-demand analysis of available nodes and interconnection networks, similar to the Network Weather Service [237] of the AppLeS project.

This model relies on maintaining a mean size for objects of given types, in order to make future estimates. Since the same tests are repeated during performance analysis, we are unable to take advantage of the repeated creation of objects of the same type. Consequently, the estimation effects are unable to be seen in these performance results.

A serious limitation of the current implementation's performance is due to Java's virtual machine (JVM). The amount of memory that a Java virtual machine can use is limited by the amount of physical and virtual memory on a node. This places an upper limit on the amount of data that can be stored in the DISCWorld daemon's store in the current implementation. It was observed that when the JVM runs out of memory, not only will it eventually crash, but the daemon fails in curious (but unpredictable) ways: some threads will be created but not others; partial parsing of new processing requests will be performed; and, some requests for information in the store will be successful, others will not. The presence of other users' processes running on the nodes further reduces the amount of memory that the daemon can use for caching.

When a service is moved to a new node, the execution time and variance are assumed to be the same as that of the node from where it has come. When the service is downloaded to a node, the past run-times are set to zero, so new averages and variances can be computed. Thus, the scheduling node makes the assumption of the same general performance characteristics; when the service is used the new values replace the assumed characteristics.

Due to the way in which the `mpeg_encode` program is implemented, each of the images to be used for input must be written to disk. This is primarily so that the images can be converted to a file format suitable for the program. The current implementation of the createMPEG service writes the files in the base format required

by the program. Unfortunately, the need to write files to a disk makes the service reliant on the amount of space available on the disk. If, during the course of service execution, the disk fills, the service fails. While this does not cause the daemon to crash, the user client is not notified, and no attempt is made to rectify the failure.

A simple timer thread is used to approximate the load on a node. The operation of the timer thread is very simple: it resets and increments a counter for a given time period. Depending on what percentage of the current maximum is returned by the counter will give an estimate of the load on the node. In the current implementation, services are only executed once the estimated load is below a certain threshold. Of course, the execution of services is not the only contributing factor to the observed load: the parsing of new DJPL scripts; any blocking requests; and saving the state of the store all contribute significantly from within the JVM. This figure will be additionally affected by the external load on the system. In future versions of the daemon, a more sophisticated method of estimating load must be developed and incorporated.

The daemon's thread count, shown in figure 46, shows at least 8 threads that are executing while the daemon is quiescent. This figure can more than double when processing requests are being parsed, distributed and executed. While implementations that use *native threads* can handle the large number of threads created during processing, we have found that implementations that used *green threads* had serious synchronisation problems. In fact, although the daemon was developed on Java 1.2 on Windows NT 4, the release of Java 1.2.1 for Solaris was the first in which our implementation did not freeze.

As the DISCWorld daemon (and client program) are written in pure Java, we must rely on the mechanisms provided within the Java language specification [88] and the Java Runtime Environment [218] for robustness and fault tolerance. Although the prototype emphasises the usefulness of adaptive scheduling and DRAMs as an enabling mechanism, for the remainder of this section we discuss how reliability, robustness and fault tolerance might be incorporated into a production version of DISCWorld written in Java.

One of the fundamental issues that must be addressed in a production system is that of availability. On whatever platform the daemon runs, there must be a guarantee that whenever a host receives a message destined for a daemon, a daemon is running to accept it. On a UNIX machine the DISCWorld daemon can be treated similarly to the operating system daemons. An entry can be placed into the file `/etc/inetd.conf`

so that if a message is received on the port dedicated to the DISCWorld daemon, a new daemon will be created if none is currently running. Similarly on Windows NT a file in `/WINNT/system32/drivers/etc` can be modified to the same effect.

As implemented, the DISCWorld prototype has two major failings: the store fills very quickly, which causes the daemon to crash; and, if the daemon fails while parsing or executing a processing request, the request is forgotten when the daemon is restarted. Problems with the store and avenues for future research are discussed in section 8.2. An obvious solution to the problem of fault tolerance is to store all daemon state information in a Java-accessible object-oriented database such as Oracle [182], Informix [121] and JDBC [218]. Thus by using transaction technology to ensure the database remains in a consistent state, when the daemon is restarted the state is able to be restored.

Writing all state information to a database could result in an unacceptable level of system overhead. Thus one of the management policies enforced by the daemon might be to categorise all state information and only store the highest levels. This scheme, similar to an incremental file backup system, would ensure that when the daemon receives important messages such as processing requests or DRAM dereference instructions, they will be stored. If the daemon is restarted after the message has been saved the action can be re-performed without too much cost to the daemon. Network failures are harder to detect because of the lack of acknowledgement messages in the daemon-daemon and client-daemon protocols. We are investigating the use of generous time-outs after which messages are resent.

Inside the daemon Java's strong typing and byte-code verification system provide the necessary framework within which a type-safe environment can be built. Other researcher in the Distributed and High Performance Computing group are working on adding user- and message-verification functions into the daemon's portal module. Authentication systems such as provided by Kerberos [173] are being considered for this task.

7.4 Conclusion

In this chapter, we have analysed the performance of the prototype DISCWorld daemon. The experimental results have shown that the scheduling algorithm is practically useful and DRAMs are a good enabling mechanism. However, the performance of the daemon is limited by the fact that several crucial parts of the

daemon have not been implemented. The limitations of the current implementation are discussed, which suggest a large body of possible future work in this area.

We have compared the performance of the DISCWorld daemon with Java RMI and cgi-bin versions of the same programs. Results have shown that while the DISCWorld daemon is slower than the RMI version in the construction of the same data products, the time taken for the RMI version is constant, whereas the DISCWorld daemon benefits from caching all intermediate results. The DISCWorld model of computation also allows client-side computation, which is unavailable in cgi-bin scripts.

DRAMFs allow demand-driven execution of processing requests. Execution of processing requests can be optimised if the same data is available from alternate nodes. The prototype DISCWorld daemon has been shown to have performance which scales linearly within the limitations of the current implementation. A detailed example has shown how DRAMs can be used to construct and execute complex processing requests that span multiple servers. An event timing diagram has shown the relative time costs of the different processing stages of request execution.

It is important to remember that although the DISCWorld daemon is a vital component in the implementation of scheduling in DISCWorld, the creation of a perfect DISCWorld daemon is not the aim of this thesis. The main topics are: the creation and distribution of good service placement decisions in the presence of incomplete system state information; and, the implementation of a framework that allows high-level information about data and services to be traded between participating DISCWorld daemons and clients, to allow on-demand client- and server-side processing.

We have achieved these two goals through the use of: a platform independent Distributed Job Placement language, with which processing requests can be described; an execution-cost minimisation function, which incorporates the concept of a daemon's willingness to perform a given function; and the DISCWorld Remote Access Mechanism, which allows services and data, whether the data has been physically created or not, to be accessed and traded between clients and servers.

Chapter 8

Conclusions and Future Work

The target audience of this work are users that need to pose high-level processing requests to a metacomputing system without need for detailed knowledge of the design and implementation of the system. We cannot assume that users will be aware of the best way in which their request should be structured, or where the components of their decomposed processing request should be placed to achieve optimal performance. In order to provide users with a simple, high-level view of the system, we abstract away from the heterogeneity of the system's components (including processors, interconnection networks, data and program code sources).

We have developed and presented a scheduling model and placement algorithm (in chapter 5) to select the most appropriate location for the components of a processing request to be executed. We have developed a mechanism for remote access to data and high-level program components or services (presented in chapter 6). An implementation of this mechanism was discussed within the context of our prototype metacomputing project, DISCWorld. The performance of our implementation was analysed in chapter 7.

8.1 Conclusions

Chapter 2 of this thesis presents a brief introduction to cluster computing and a review of current metacomputing projects. We conclude that while very good solutions to this problem exist, most cluster- and metacomputing environments target those users with a parallel or distributed computing background. Users without such knowledge, wishing to pose high-level processing requests to such a system, are not catered for. We address such users in the Distributed Information Systems Control World

(DISCWorld) environment. Our system is characterised by the use of re-usable high-level program components and movable program code and data. We provide infrastructure to allow the incorporation of programs implemented using pure Java, Java Native Interface (JNI) or system calls to legacy software.

We critically review the systems in chapter 2 with focus on their resource management capabilities. We conclude that, with few exceptions, most of the environments reviewed use *ad hoc* scheduling methods. Other metacomputing systems provide no true scheduling support, but instead allow resources to be reserved by users and let the resource's local scheduler take control of program execution.

Furthermore, our review of the scheduling literature in chapter 3 shows that while the problem of scheduling parallel and distributed programs has been extensively studied, there has been little work that considers both the data and program code to be movable, or semi-movable. In addition, most of the literature assumes complete system state information and that results of previous programs are not useful to other programs. Complete system state information is difficult to maintain. We believe that partial system state information *can* be used to good effect. Nearly all scheduling research considers the system to be controlled by a centralised scheduler. We conclude that while the use of a centralised scheduler allows good load balancing between processors, it presents a processing bottleneck.

We conclude that there is no scheduling model that exactly fits the characteristics exhibited by our system where: data and program code are both movable to restricted processors; hybrid static-dynamic scheduling is used (which allows for runtime optimisations); decentralised scheduling is integral; partial system state information is exploited; and where a global naming mechanism for data and program code exists.

In chapter 4, under the assumption of partial system state information, we explore the effect of different static and dynamic scheduling algorithms on mixtures of independent programs. We show that when the execution time of programs are drawn from different real-world distributions, there is *no one* scheduling algorithm which produces the single best execution schedule. We conclude that an *adaptive* scheduling algorithm is necessary when jobs exhibit different non-trivial distributions of execution time. The tool developed in chapter 4 enables adaptive scheduling across a network of workstations, and can also be used to perform parameterised numerical simulations under the control of DISCWorld.

In chapter 5, a model for scheduling and a heuristically good placement algorithm is developed for use in metacomputing systems with partial system state information.

The model addresses the characteristics of our system (as detailed above). We conclude that because of the federated nature of the resources that comprise a metacomputing system, an *adaptive* scheduling mechanism is required to make use of partial system state information. The model allows optimisation of execution schedules through the re-use of both final and intermediate results from previous jobs.

We also develop a mechanism by which a high-level job's internal structure can be represented, and processing requests controlled. A job's internal structure can be represented by a directed acyclic graph (DAG). We formulate this using extended markup language (XML). Our language, termed the *Distributed Job Placement Language* (DJPL) is used to express a DAG in textual form. Consequently, requests are transmitted in an architecture-independent manner. The DJPL contains enough information to allow any node in the distributed system to create an execution schedule for the processing request. Program code and data are referenced by their global names. When the processing request is scheduled, an *annotated* version of the DJPL is used to distribute the request to the nodes selected for execution. The DJPL allows the delegation of processing responsibility to other nodes in the distributed system.

In chapter 6 we show how this scheduling model and placement algorithm can be implemented using the features of DISCWorld. We conclude that a remote data pointer is necessary, but to prevent the problem of dangling pointers arising, should rescheduling be necessary, we develop the DISCWorld Remote Access Mechanism (DRAM). Since DRAMs embody the recipe for reconstructing the data they point to, should it be unavailable, DRAMs cannot dangle by definition. At worst, a short delay is introduced into a compound processing job, to reconstruct the data.

In the course of constructing DRAMs it became apparent that a worthwhile extension, the DRAM *Future* mechanism (DRAMF) would allow a lazy creation of the data to which the DRAMs pointed. This allows yet further optimisations in the scheduling and placement, with data products being created "on demand", or just in time.

Our prototype implementation of DISCWorld, which incorporates just enough infrastructure to experiment with the scheduling and placement mechanisms described in this thesis, performed better than expected. It embodies the DRAM, DRAMF and DJPL components. In chapter 7 we showed how a set of services could be implemented to reproduce the behaviour of the image archive management system

described in appendix B. Although, as expected, DISCWorld introduces some startup overheads, it compares favourably against the cgi-bin based system and illustrates the value of a data product caching mechanism.

8.2 Future Work

The work presented in this thesis has generated a number of ideas for future research. Improvement of the store, described in section 6.5.3, is the most critical aspect of future work to arise out of this study. There are two fundamental limitations introduced by the current store. Firstly, because the store is implemented as a Java vector, any exceptions caught while reading the vector cause the read operation to be terminated. Secondly, because all results and partial results are cached in memory, the store quickly fills, exceeding the the memory capacity of the Java virtual machine.

The store is implemented as a single data structure (a `java.util.Vector`). When an object's class file is changed (which produces a new serialization ID), the saved vector becomes invalid. The current DWd implementation reads the serialized vector until an object is found with an invalid serialization ID, and truncates the remainder of the vector. The problem of transparent version control is an issue which must be addressed in a production version of DISCWorld.

When the store uses all the memory allocated to the Java virtual machine, the daemon crashes. The limitation of memory can be addressed through the use of a high-watermark and low-watermark [155] management scheme for the store. In the DISCWorld architecture model, one of the Manager's functions is to maintain information about the point at which old data is removed or backed up to permanent storage (the high watermark), and the point at which data may be retrieved from the backing store to speed up access time (low watermark). We are considering implementing the store as an object-oriented relational database (OORDB).

In addition to the removal of the DWd's storage limitations, we are considering the addition of economic cost models into the scheduling model and placement algorithm. We have made provision for costing model information in the prototype DJPL specification, presented in chapter 5. We anticipate users will specify time limits by which their request should have finished executing. The system allows users to trade cheap slow services against expensive fast services. Cost may be expressed in some system-wide currency unit.

We are also considering the case in which services are duplicated to achieve better execution time. This is discussed in [205]. When coupled with an economic model, this introduces a trade-off between the level of service duplication and the amount that a user is willing to pay. The localised delegation mechanism provided by DRAMs can be used to implement duplicated services.

The cancellation and modification of schedules is another important area of future work which we can now address. When a user cancels their processing request, the behaviour of the system depends upon the point at which the cancellation is made. If the user wishes to cancel their request *after* the submission of the DJPL script but *before* they inspect the output DRAMs, then there is no further action taken by the system. The DRAMs will continue to exist, and may be cached and traded by the DISCWorld daemons, as normal data objects. If, at some later stage, a user client or daemon wishes to inspect the DRAM's data, this will either cause the data to be retrieved from the source, or to be created in accordance with the DRAMF's recipe.

The recent release of Sun Microsystems' Jini technology [15] allows applications to be packaged as services that are available across a shared tuple-space, JavaSpaces [75]. We are investigating the possibility of using JavaSpaces for inter-daemon communications and for resource discovery of not only daemons, but services. We believe that Jini and JavaSpaces have the potential to significantly reduce the number of challenges faced in metacomputing.

The work presented in chapter 4 can be extended into a framework in which the performance of jobs executing on slave hosts is used to build up user job distribution patterns. Using the conclusions from chapter 4, the distribution could then be used to properly select a near-optimal static or quasi-static scheduling algorithm. A further problem is that of the framework's sensitivity to new job whose execution times are outwith the previously-derived distribution. A further problem is thus when to rebuild the distribution in order to adapt to new jobs. If the framework is executed as a DISCWorld service, DRAMSs provide place-holders for conveying jobs' characteristics, such as the mean and variance in runtime of a service (as described in chapter 6).

The possibility of interaction between DISCWorld and other metacomputing environments such as Globus and Legion is a logical extension of this work. At a high level, both Globus and Legion may be accessed via the DISCWorld service wrapper. That is, existing Globus and Legion applications may be encapsulated within a service wrapper and executed by the DISCWorld daemon directly. In order to facilitate this

style of interaction the Globus or Legion run-time systems would have to already be executing.

From a lower-level viewpoint Legion's persistent object space might be used to store and execute services (neglecting the fact that DISCWorld is implement as pure Java and Legion is implemented in the Mentat programming language). Once differences between the DRAM and Legion object interfaces were resolved, DRAMs could be directly used by the Legion system. The fundamental difference between the two systems is that Legion is designed for fine-grained parallelism whereas DISCWorld is targeted at coarse-grained distributed computing – the emphasis is not on parallelism.

There are two possible avenues for DISCWorld to interact with Globus at a low level. The first is for the DISCWorld client program to be executed as part of a Globus user query. This would involve the user specifying that they wished to use the DISCWorld as part of the processing commands. When their request is refined into RSL one of the machine host criteria might be that the machine run a DISCWorld daemon. Thus an appropriate entry would have to be added into the LDAP schema used for selecting Globus machines. Conversely, DRAM objects sent from a DISCWorld daemon could quite easily be accepted by other programs associated with a Globus user request. Because the granularity of the systems is so different other difficulties may be found. For example, in DISCWorld only daemons may send and receive DRAMs. In Globus each program participating in a user request handles its own message communications.

Other quasi-metacomputing system, such as Nimrod could be quite easily incorporated into the DISCWorld environment using a service wrapper. That Nimrod utilises distributed processors to perform task-granularity parallel computations is inconsequential to the operation of the DISCWorld daemon.

In conclusion, we have shown that while an adaptive scheduling mechanism requires a significant amount of supporting infrastructure, it *can* be usefully implemented in a metacomputing system. We have found Java to be a convenient implementation vehicle and have developed some useful modules which we believe are sufficiently general to be of use in their own right.

Appendix A

Distributed Geographic Information Systems

A.1 Introduction

In this chapter we discuss the issues involved in implementing a distributed system that provides a computational infrastructure for the development of decision support and research applications requiring access to and manipulation of large geospatial data sets, such as satellite imagery, from remote servers. We give an overview of some prototype systems we have developed, and some more advanced systems that are still under development, which employ standard Internet and World Wide Web client/server technology. This chapter provides a summary of [43] and [112].

The amount of digital geospatial data available is rapidly growing. In particular, there is a vast amount of data from earth observation satellites, and next-generation satellites are expected to produce terabytes of data per day. This presents a challenge for the development of computer systems that enable the storage, management and dissemination of these huge data sets in online data archives or digital libraries. Ideally, such a system would provide efficient, on-demand remote access to these data sets over the Internet (or an intranet), so that authorised users could easily access and utilise the data for a variety of Geographic Information Systems (GIS) applications, including decision support, research and other analysis.

For a number of GIS applications, such as those requiring real-time or interactive analysis of large data products such as satellite imagery, the processing requirements are large enough that high-performance compute servers are required. This leads to the concept of an "active" digital library, where the server provides not only services

for querying and downloading of data from the library, but also services for processing the data before downloading [106, 108, 113, 127, 129, 235].

This approach is particularly useful if the amount of data to be processed is very large, for example multiple channels of a satellite image, but the final result is relatively small, for example a processed satellite image for a localised area, or perhaps just a few numbers such as average sea temperature or percentage cloud cover or some correlation coefficients. If the data is obtained from the server using a wide-area, relatively low-bandwidth network, it will be more efficient if the user only has to download the final results rather than download the large input data set and process it locally. Many decision support applications that manipulate spatial data involve operations on very large data sets, but carry out data reduction operations to provide summarised information to the end user. Some of the data sets may be remotely accessed from different servers, possibly over wide-area networks, and the processing may be done on yet another machine, possibly a high-performance computer or supercomputer. We have investigated the consequences of connecting together resources for fast mass storage and high-performance computing with broadband networks, whereas the user's client computer may only have modest network capabilities, such as a modem link via the World Wide Web.

We have experimented with large image repositories such as for geostationary meteorological satellites, as well as other sources of geospatial data such as digital terrain maps. Operations on bulk data range from simple data overlays, to computationally intensive operations such as data interpolation and registration and rectification. Some of these operations are infeasible to do in real-time except on supercomputers. The client programs that can set up processing on demand of various data sets need to be robust, and yet easy to use with suitable graphical interfaces.

In section A.2 we discuss mechanisms for enabling an active digital library, using as an example our online repository of geostationary satellite data and some prototype systems for providing access to this data and services for processing the data. We discuss the issues we have encountered in implementing prototype systems using Web, Java and CORBA technologies. In section A.3 we present some general scenarios illustrating how decision support systems can benefit from the use of fast on-demand access to archives of geospatial data and distributed GIS services, with more specific applications being described in section A.4. We briefly summarise our findings and conclusions in section A.5.

A.2 Online Geospatial Data and Services

With the advent of cheaper and more powerful computers, networks and electronic storage media, and particularly the huge increase in the use of the Internet and the World Wide Web, large online data archives and digital libraries that can be accessed over the Web are becoming widespread in both the commercial and scientific arenas. The technologies for implementing and accessing such archives, are for enabling online data processing, are still evolving. Our On-Line Data Archives (OLDA) Program [178] is investigating these issues, particularly as they relate to large satellite data archives.

There are a number of existing projects aiming to provide access to digital libraries of earth observation data. These include the Synthetic Aperture Radar Atlas (SARA) at Caltech [234,235]; NASA's Earth Observing System Data and Information System (NASA-EOSDIS) [164]; the European Space Agency's online product catalog [60]; and the Australian Centre for Remote Sensing (ACRES) Digital Catalogue [17].

Like these and many other projects, we are investigating the computer systems and software issues related to the efficient storage and dissemination of data in large distributed online archives, or digital libraries, of satellite data. However our work, and similar efforts such as the SARA project, is also focusing on developing active digital libraries, which enable remote data processing as well as remote data access. New technologies such as the World Wide Web, Java [88] and the Common Object Request Broker Architecture (CORBA) [163, 177] are making it easier to develop portable distributed client/server systems of this kind.

We are implementing our software support infrastructure using a set of server-side programs accessed using a mixture of Java programs communicating between client and server as well as between servers; CORBA for invoking remote procedures through an object-oriented interface; and the Common Gateway Interface (CGI) [100], the standard mechanism for invoking processes on a Web server. The client is a customised Web-based interface using Java applets [88] that can be downloaded and run from a standard Web browser. This architectural mix provides a portable and powerful framework for rapidly prototyping systems that can integrate existing server-side utilities, and is extensible to create complex systems with the desired functionality. Most importantly, it is built using Web standards, with standard Web browsers for the clients. Our prototype system, Eric, implements an active digital library browser and manipulation tool for spatial data. Eric is discussed further in section B.

A.2.1 Standards and APIs

In order to make Web-based distributed geographic information systems more general, so that the clients and servers can handle any kind of geospatial data set, and to develop distributed systems with multiple clients and servers that are interoperable, there are two important standardisation issues that must be addressed.

The first is the standardisation of metadata, so that searching digital libraries by querying metadata can be done using the same metadata fields for different data sets. There are several organisations working on metadata standards, both for specialised fields such as geospatial data [125, 222, 223], and in general [51].

The second issue is the standardisation of the client/server interface, so that different clients and servers can be interoperable, and clients can request data from multiple repositories using a standard interface, or an application programming interface (API). The API to an active digital library should specify mechanisms for querying, processing, and retrieving of data. Using general metadata standards, it would be possible to construct a basic general API for querying of metadata and downloading of data that would be universal across any kind of digital library and any kind of data (general Web search engines do this for restricted data sets, such as Web pages and newsgroup postings). However to allow more powerful and domain-specific queries, and to provide an interface to data processing services which would in most cases be specific to the application and type of data, specialised APIs are needed that are specific to a particular domain or topic — for example, the services and interfaces for an active digital library of geospatial data would be different to those for a library of human genome data.

To develop a more advanced active digital library for geospatial data, integrated with an improved version of ERIC, a well-defined API is required. As with metadata, proposed standards are gradually being developed for digital libraries and data archives. There are various groups, including the Open GIS consortium [179], the International Standards Organisation (ISO) geographic information technical committee [125], and various government agencies in the U.S. and elsewhere [223,229], that are all involved in developing standards for the storage and exchange of geospatial data.

Currently the API that appears to be the most advanced in design, and the most suited to our existing archive of GMS-5 satellite data, is the Geospatial and Imagery Access Services(GIAS) [228] specification, which is a fundamental part of the U.S. Imagery and Geospatial Information System (USIGS) [226, 227] developed by the

U.S. National Imagery and Mapping Agency (NIMA) [229]. This system is targeted at use by the U.S. military, for analysis of satellite and photo-reconnaissance imagery, however it is general enough to support the storage, management and dissemination of geospatial imagery data for a more general distributed GIS. We are therefore developing an active digital library for GMS-5 satellite data that is based on the GIAS. This prototype system could be generalised to support any kind of geospatial data.

A.2.2 Implementation of a Standardised Geospatial Data Archive

The GIAS specification describes an object-oriented archive management system for an active digital library. The specification is structured so that the main functions of the library, such as adding data, querying the metadata, and downloading the data of interest, are controlled by different *managers*, each of which is handled by a different class. Remote access to the system is implemented using the CORBA standard, and described using Interface Description Language (IDL) [163]. The GIAS also defines a standard Boolean Query Syntax (BQS) which specifies a format for metadata queries on geospatial data. More detailed information on the GIAS can be found in the specification [228].

We have developed a prototype implementation of a GIAS-compliant geospatial digital library [45]. This work was done in collaboration with the Imagery Management and Dissemination (IMAD) Project group of the Australian Defence Science and Technology Organisation (DSTO), who are implementing a prototype distributed system for handling photo-reconnaissance imagery for military command, control, communications and intelligence (C^3I) applications. The same system can also be used for a variety of non-defence GIS applications, such as those described in section A.4.

We have initially implemented a subset of the GIAS that provides the basic functionality for managing a geospatial image library. The implementation includes the main managers for the server, an interface to a database on the server for storing the queryable metadata, a translator that converts BQS queries to standard SQL database queries, the remote invocation of the managers using CORBA, and a client for testing the complete functionality of the subset of the GIAS that we have implemented. Figure 49 shows a screen image of the test client. Only the queryable

metadata is stored in the database, while the imagery data is accessed via the standard file system, and may be stored either on disk or in a tape silo.

The GIAS provides a sound basis for implementing an active digital library of geospatial data. It specifies interfaces to the basic functionality of searching for data products by querying metadata, and downloading the required data in a specified format and resolution. It also provides interfaces for some basic data processing services of the kind provided in the original ERIC system, such as cropping out a specified geospatial region, and creating animations of time series of images. The design, which is based on CORBA, is capable of supporting additional data processing services, however an API would need to be defined for each one, and included into an extended interface specification.

We are currently working on interfacing our existing GMS-5 repository to the GIAS server. This requires loading metadata for all the existing GMS-5 data into the database used by the GIAS implementation, and converting our automated data ingest programs so that new data from the satellite is automatically incorporated into the GIAS system.

The GIAS implementation provides most of the services required to support the functionality of ERIC. We are developing a new, GIAS-compliant version of ERIC, using a Java applet rather than HTML forms as the user interface, connecting to the server using CORBA rather than CGI, and implementing the server programs using Java processes that can call other programs as native methods, rather than calling them from shell scripts and Perl programs. Using CORBA requires an Object Request Broker (ORB) running on the client as well as the server, however CORBA and the Internet Inter-ORB Protocol (IIOP) [163, 177] that it uses are now being integrated into Web servers and browsers, for example the latest versions of Netscape Communicator feature a built-in Java ORB to handle CORBA requests [168].

By implementing the ERIC user interface as a Java applet, it can easily be customised for different applications, which can download the basic applet, plus classes specific to the particular application. Using Java programs on the server allows for a more complex and modular system than could be constructed from shell scripts and Perl programs, and allows much better error handling, which was a major problem with the original version of ERIC. The GIAS defines standard exceptions (errors) as part of the specification.

A.3 Distributed Systems for Decision Support

In this section we illustrate some typical scenarios for decision support systems that access distributed data archives across wide-area networks, and discuss the middleware and distributed computing issues which we believe are critical for their widespread uptake.

One example from the area of land and environmental resource management is particularly relevant to Australia, which derives a substantial part of national productivity from primary industries. A great deal of spatial and land resource information is now available from various satellites, aerial reconnaissance, measurement study programs, and other sources such as census and land registry. There are many interoperability and ownership and legal access issues involved in multi-source data. Although these are a great impediment to wide-area distributed decision support systems, we do not address these issues here. Another significant impediment is the difficulties in setting up the software interoperability and machine access for decision makers to access and manipulate such data. We will concentrate on these issues, and in particular discuss the consequences of wide-area distributed computing and metacomputing in addressing these difficulties.

Imagine a station manager or farmer planning the year's activities and wishing to make sensible decisions regarding irrigation and rainfall runoff; crop rotation and planting; expected yield and optimal harvest times and other land care operations such as salinity management and prevention. It is unlikely that the farm manager is working without some existing base of knowledge. There is very likely a base of experience that may have been passed down in the family or is available from neighbouring stations or other individuals in a similar situation. Nevertheless, there may be newly acquired land or new situations, new crop species or irrigation techniques that expand the list of option available to the farmer. How can they exploit the data that may be available to aid the decision making process? Some data will be available in the form of highly sophisticated processed data products such as short and long term weather forecasts. These may be available at the resolution and localisation required for precision agriculture decisions, or may only be available in undigested form. Perhaps no-one has asked for detailed forecasts or data for the farmer's particular region before at the requested resolution. It may however be entirely possible for an automated processing system to be able to create the desired products. Aerial reconnaissance data can be readily bought for a particular region by hiring a suitable plane and aviator. Fly-over data at very high resolution from

various commercial satellites is quite likely available in raw form for the region the farmer requires. How can a processing and automatic product creation framework be put in place to allow those organisations who do have the necessary skills to create the desired products to do so economically for what might be a set of isolated one-off sales?

A wide-area metacomputing environment built using a set of clustered computing resources can be set up to provide a common shared resource for the customers (farmers and land managers) to interact with the value-adders or organisations who can create processed products from raw data and the raw data suppliers or government custodians. What is needed is a suitable set of middleware or software that can provide the necessary interoperability and scheduling of the necessary computing services.

There is a feeding chain relationship that exists between the end user of a processed product, the custodians of the raw geospatial data, value-add processing agencies which may involve several intermediate stages, and the final data suppliers. In many cases around the world raw data belongs to the government and may be made available for the public good to anyone who wishes to use it. Government agencies typically have a dedicated and customised infrastructure for processing raw data to construct decision support products. Government operated weather bureaus are an example of this. The feeding chain model for this is shown in figure 50, where the entire system can have dedicated hardware, software and a high bandwidth data access system.

Many organisations now exist to sell products derived from this raw data, possibly involving considerable expertise and innovation as well as sophisticated processing equipment. The feeding chain model that allows vendor companies to value-add to existing data is shown in figure 51, illustrating how the end-user receives the final product through the vendor alone and need not be aware of the raw data such as satellite imagery that went into its construction.

Some value added products such as weather forecasts are by no means trivial to derive from raw observational data, and weather agencies around the world expend a lot of resources in assimilating data and producing their products. The economics of forecasting is selling to a large volume market, since so many people and organisations want accurate weather forecasts that they are generally sold at a very low margin, if any at all. Land management decision support products tend to be more esoteric and require large margins to be worthwhile for a value-adding organisation. In consequence the decision support capability of such data tends to be underutilised.

Many government agencies set up their own special systems (as shown in figure 50) to provide spatial or geographic decision support for their areas of jurisdiction. Weather agencies are moving towards more automation which will lead to a driving down of the cost of production of value-added forecasts, but automation is even more necessary for low-volume market areas.

The low-volume market can be effectively served by a hierarchy of different value-adders all interoperating, on-selling products derived from each other's work, and ultimately derived from the bulk raw data that may itself originate from satellite or other sources run by a commercially-operated company. This hierarchical relationship is shown in figures 51 and 52.

A number of general purpose computing technologies can be used to improve the automation of these value-added services. Improved data archive management facilities can provide faster bulk access to data sources more cheaply. A number of systems for archiving data make use of what is essentially a smart middleware to control a mix of storage media – many cheap bulk devices such as tapes are housed in multi-tape silos under robotic control, working in tandem with fast but more expensive devices such as bulk disk or RAID arrays of disks which can act as a buffer for frequently accessed data. The primary data suppliers may choose to use technology such as this in managing the bulk archives of raw unprocessed data. Value-adder organisations may require access to more than one primary source of bulk data simultaneously, as shown in figure 53, and it may prove economic for suppliers to use on-line archive technology such as combined RAID and tape silo systems to minimise the amount of data that needs to be replicated at various sites.

Processing resources can be pooled together by organisations to provide better response time or resource utilizations than if individual departments under-utilised their own resources most of the time just to be able to handle their occasional peak capacity requirements. In particular many organisations have clusters of workstations or personal computers that are only sporadically used by individuals during working hours and which stand idle overnight and on weekends and holidays. It is often too difficult to make use of these lost cycles due to lack of smart interoperable scheduling software environments. Improved scheduling on clusters can turn an unused resource into a virtual supercomputer for these organisations.

Storage technology can be integrated with World Wide Web data delivery techniques to provide casual customers access to a plethora of pre-prepared data products with very low shipping and delivery costs in spite of up to the minute

relevance. Nevertheless public demand for network bandwidth is growing and better data connectivity between data suppliers and value adders is likely to be necessary to avoid competition with public Internet traffic. Smart middleware can be used to manage the existing networks between cooperating suppliers in a feeding chain, to make sensible use of data caches and intermediate storage so as minimise peak loads and hence conflicts on the networks. This idea is illustrated in figure 54 where a value-adder has some sort of smart store that can cache products according to some access policy and thus archive processed data. A smart middleware management system is capable of managing access pattern predictions and pre-fetching data for certain applications and users. This would make use of a network of caches and archives to make optimal use of the customer data delivery network.

End-users of a particular value-adder can generate a homogeneous or heterogeneous mix of product requests. A particular vendor may benefit from the caching model shown in figure 54 even if the requests are heterogeneous in nature, providing the intermediate products, from which different final products are derived, are cached to allow fast reuse.

In addition to the normal feeding chain of products supplied by a value-adder to their customers, some degree of product development is likely to be ongoing as an in-house effort or perhaps in collaboration with special customers. These new or one-off data products may be researched and developed by special users who have access to the individual processing services inside the value-adder's middleware system. The decision support products thus developed may eventually make the transition to being fully automatable products for normal on-demand delivery to all customers. This model is shown in figure 55. The value-adding organisation's staff then spend their time developing new products and services, and encoding these into the smart scheduling environment as well as handling those occasional special jobs which require some manual intervention. The schedule management environment might still provide substantial help to the value-adders staff undertaking special jobs, as it can provide the primitive operations and support environment. We now have a mixed scenario with different levels of users all drawing different services from the environment.

In summary, better middleware to manage storage of raw data, exploit under-utilised processing capabilities such as existing clusters of workstations, and improve data network delivery using mechanisms such as smart caches and pre-fetching, can significantly enhance the performance and cost performance perceived by decision support users. This is particularly effective over large wide-area clusters of resources,

where one might expect the statistical load fluctuations from individual users is balanced out.

A number of middleware products have already been attempted and several techniques are being presently researched. An area we are addressing ourselves is that opened up by restricting the general scheduling problem to that of handling only well-defined high-granularity services.

Consider the types of services that might be required for the land care and management decision support scenario we have outlined. These are not arbitrary programs running with arbitrary data. Instead they are generally drawn from a subset of well-defined application components with known performance requirements and will typically be run on a very restricted set of data sets and data set sizes. Consequently it is possible to set up a scheduling management system that has access to the characteristics of each application component and can therefore predict and hence optimise more accurately the cost and time needed to carry it out on a set of computing resources. The smart scheduler is therefore able to make very good use of the whole set of resources under its control and can organise user requests in priority, running the most important or urgent on fast resources, and the lower priority actions using slower, cheaper queues. Given that a restricted finite or manageable set of applications and data sets are being used under the system as a whole it is also possible to effectively cache or store frequently accessed or requested data products.

Consider an example. Suppose a farmer has requested information regarding the prediction of crop yield and optimal harvest time. Crop acreage may already be known to the farmer, or can be accessed from either a land registry database or perhaps calculated from a satellite image which is geo-rectified and registered to allow a geographical area calculation. Localised weather data for the current growing season, weather predictions for the region, and weather data and crop yields from previous years may be combined to calculate an estimate of the likely crop yield. A practical system might provide a series of possible harvest time predictions and options with the computed consequences for the farmer. The decision support product delivered to the farmer may only be the summary output of the calculations – a few kilobytes perhaps, whereas the raw data upon which the calculation was based may have represented many gigabytes of spatial imagery, digital terrain data, runoff patterns and so forth. The final product can therefore be easily delivered over a low-bandwidth network to a client application running in a Web browser.

Similarly, the computational power of the client computer the farmer might use to

place the query and request the decision support product might be low, whereas many processing cycles may have been used by the value-adding organisation in creating the data product. Suppose the farmer makes a similar request the next day, but perhaps with some additional new information. By controlling and caching the intermediate data products the value-adding organisation used to create the product such as the raw satellite imagery for the farmer's particular region, it may well be possible to save considerable re-processing time. The profit margin on the second product will be much greater - or of course the value adder may choose to pass on the margin saved to its customers in the form of a cheaper price. This model of being able to cache intermediate data products may provide significant savings and better utilizations of resources. It can only be set up under a smart caching system however. It would be too hard to track the necessary data items manually.

The farmer's query is decomposed into a sequence of well-defined and computable steps in the example above. This is shown in figure 56. Each of these steps might be an application component with a well-characterised performance that can be controlled by the smart scheduler. The value-adding organisation that provides some particular services and data products might have access to archived raw data either as a locally archived copy or from some other data provider such as a government agency or another vendor. Our value-adder might be able to respond semi-automatically to the incoming query and deliver the product and a bill to the farmer using a suitable scheduling environment which manages its computing resources.

It is not always entirely possible to automate product generation and delivery. Some products will always require some human expertise or intervention to create them. A common example is the detailed local weather forecasts issued by government weather agencies. A great deal of automatically collected raw data such as satellite imagery, and computer model generated data is used as input for regional forecasts, but the complexity of the problem is such that these data products are generally only used as input to an expert forecaster who digests the data and uses them as decision support information in producing his own actual forecast. The forecaster has final say on the final product, but has made use of a highly sophisticated decision support environment in producing it. This human intervention stage can occur anywhere along the feeding chain, and the middleware support environment can be thought of as a framework to support those humans involved.

A.4 Decision Support Applications requiring Distributed GIS

Other decision support applications share many of the properties of the scenario we have outlined above. Here we present a brief description of some of the specific applications that we are targeting with our prototype distributed GIS software infrastructure and active digital libraries.

A.4.1 Land Management and Crop Yield Forecasting

An example scenario of providing decision support information for land management using a Web-based distributed GIS was given in the previous section. There are many similar applications. For example, as well as providing localised crop yield forecasting for an individual farm, there is a demand for crop yield forecasting on a regional and national level. This requires the use of large amounts of disparate data that may be distributed over many sources. If this data is made available online, it can be integrated by a distributed GIS, which can also provide support for including compute servers and interfacing to the software to perform the crop yield prediction.

A distributed GIS has obvious applications to large-scale land care studies and environmental monitoring, which also require large amounts of disparate geospatial data, including satellite data. For example, a researcher may want to combine vegetation index data obtained from NOAA satellite images with rainfall data obtained from weather bureau data and GMS-5 meteorological satellite images, in order to correlate rainfall with vegetation growth, to provide better understanding of the environment and better models for predicting crop yields, or providing localised predictions of rainfall and frost.

A.4.2 Defence and C^3I

Many intelligence products originate from satellite or aerial reconnaissance flights and are archived using various technologies including digital data systems. Various organisations within a government's defence forces may collect and archive their own data and may make it available to each other. Sources may vary in quality and type considerably. Some organisations may work as in figure 50 in an entirely in-house customised system, but more frequently economic and interoperating reasons require various value-adding relationships to be set up with the whole defence force to share

data and to derive intelligence products from multiple distributed sources and 'on-sell' derived products for decision support. Human analysts may be working as end-users or as value-adders in the system itself, combining data products into decision support material for those processes that are not yet automatable. The defence community has some additional constraints but overall the model is very similar to that for commercial value-adding of land resource data.

Some additional properties however include: real-time or near-real time data delivery; tighter security and secrecy between components of the system; encryption and restrictions to certain parts of the 'market'; and closed rather than open product catalogues. In spite of these differences the quality and quantity of the data components and the automated and human enhanced processing activities still conform to the decision support information delivery model outlined in section A.3.

We are currently working with the Australian Defence Science and Technology Organisation (DSTO) to develop a prototype Web-based imagery access system as a technology demonstrator [45]. The system is aimed at a variety of C^3I clients, from image analysts using fast networks and high-end graphics workstations, to commanders in the field using laptops and low-bandwidth connectivity.

A.4.3 Emergency Services

There are two basic scenarios for the use of GIS by emergency services. The first is for emergency planning purposes, to investigate measures for alleviating or responding to emergencies such as fire or flood. These systems allow the user to simulate possible situations and their outcomes. For example, firefighters or forestry managers might run simulations of what would happen if a fire was started in specified areas under certain conditions, in order to identify danger areas where fuel loads should be reduced with cool burns, and under what conditions these managed burns would be safe. Emergency services might run simulations of evacuation procedures in the event of a flood or fire.

The second scenario is decision support during an actual emergency, which provides a much greater challenge to a distributed GIS. As with emergency planning, the GIS will draw on a wide variety of existing geospatial data, such as the positions of populated areas and houses, road networks, fire stations and hydrants, etc. However in this case, it will also require access to real-time data that may be rapidly changing. For example, a GIS for decision support during a bushfire would ideally have real-time data feeds coming from many different sources, including the positions of firefighters,

police and emergency services personnel in the field, information about the position of the firefronts, which can be transmitted from helicopters with GPS units, and current and predicted weather information such as temperature and wind speed.

A Web-based distributed GIS could be accessed in the emergency response headquarters, as well as by crews in the field using a laptop connected by a cellular modem. The GIS could provide up-to-the-minute information on the situation status, and provide decision support information such as the optimal routing of fire trucks or evacuations, and predicting the path of the fire using simulation.

The National Key Centre for Social Applications of GIS at the University of Adelaide is developing such systems for emergency services decision support [166]. We are working with them on addressing the complex distributed computing aspects of this application, including the access to real-time data feeds from multiple sources, and the use of high-performance compute servers to provide rapid simulation.

A.5 Conclusions

We are investigating the computer systems and software issues related to the efficient storage and dissemination of data in large distributed online archives of geospatial data such as satellite images. We are developing "active" digital libraries, which enable remote data processing as well as remote data access. Technologies such as the World Wide Web, Java, and CORBA are making it much easier to develop portable distributed client/server systems of this kind.

We are currently developing the infrastructure to support a Web-based distributed GIS. Some prototype systems have been implemented, such as the ERIC active digital library for GMS-5 satellite images, and the Imagery Management and Dissemination (IMAD) system for handling defence photo-reconnaissance images. These distributed geographic information systems, and the middleware infrastructure that supports them, are currently being provided with additional functionality and robustness. As the systems are improved, they will be used for advanced decision support applications such as land management and crop yield prediction, and C^3I applications for emergency services and defence.

Client/server computing, particularly over wide-area networks, is not yet widely used for GIS applications and research, but we believe the use of distributed geographic information systems that can remotely access and process a variety of

data from multiple online data archives has great promise in many research and decision support applications.

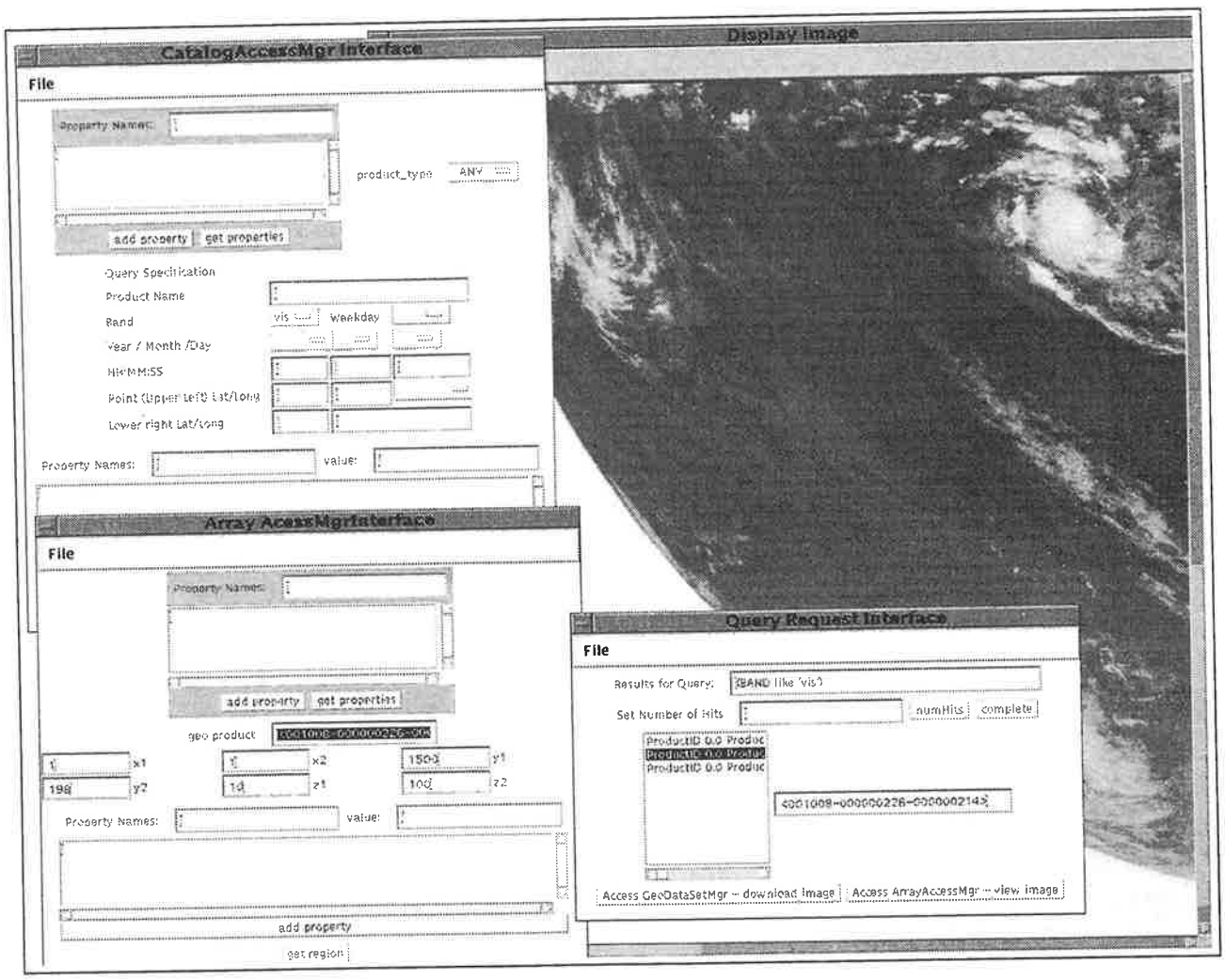


Figure 49: A screen dump of some manager windows from the test client for the GIAS implementation. In this case, a query has been made using the Catalog Access Manager, a section of one of the products matching the query has been selected and downloaded using the Array Access Manager, and the image has been displayed in a separate window. The purpose of this client is just to test the functionality of the GIAS implementation – an application client would have a more user-friendly interface.

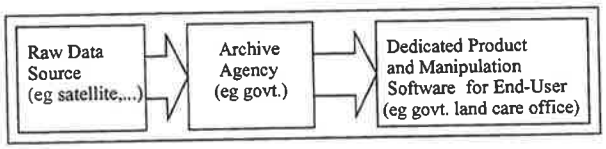


Figure 50: Dedicated decision support system for a large organisation such as a government agency (a government land management office or weather bureau for example) with its own customised, dedicated processing system for generating decision support products.

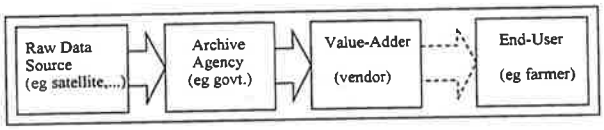


Figure 51: The feeding chain for vendors to value-add to publicly archived or government data to generate lower volume, higher margin data products for decision support.

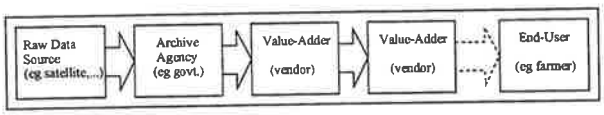


Figure 52: Hierarchical relationships between value-adders allow complex end-user products to be derived from a feeding chain of on-sold data products. Each value-adder on sells derived data products.

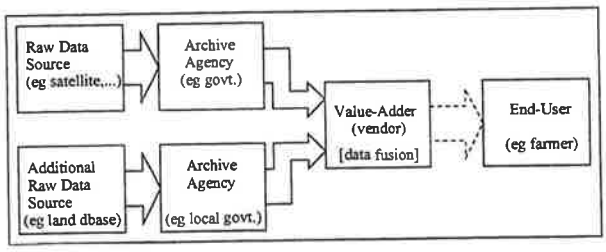


Figure 53: Value-adder combining primary data sources by data fusion into a complex output data product.

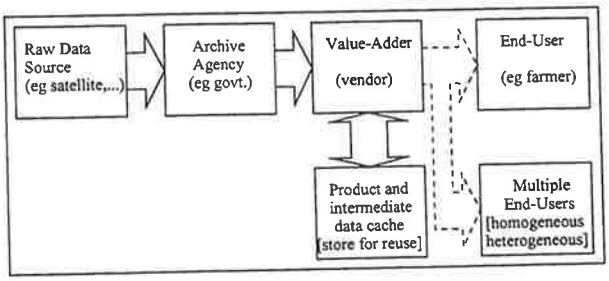


Figure 54: Homogeneous and heterogeneous mixes of end-user requests benefit from smart caching of final products as well as intermediate data used to generate final products.

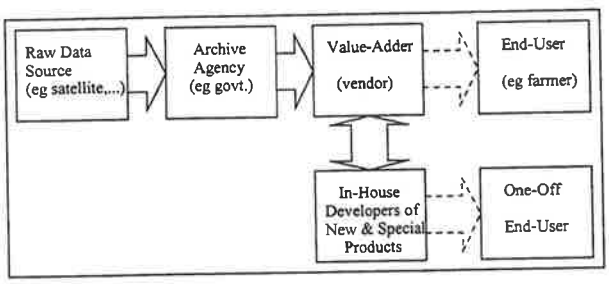


Figure 55: New and one-off special products can be developed by in-house special users, for eventual automation as on-demand end-user products.

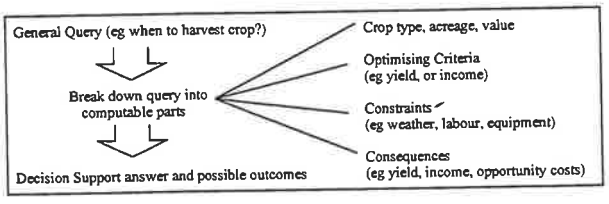


Figure 56: Decision support possibilities if a complex query can be broken down into a set of computable parts.

Appendix B

ERIC - A Simple Web-Based Satellite Imagery Browsing and Manipulation Tool

B.1 Introduction

Integrating parallel and distributed computer programs into a framework that can be easily used by applied scientists is a challenging problem. In this chapter we discuss a framework that integrates parallel processing and distributed computing software for accessing and manipulating geostationary satellite imagery. We employ World Wide Web (WWW) client/server technology to allow a universal form of user interface to our system.

Land management and environmental scientists make use of multi-channel geostationary satellite imagery in analysing rainfall and vegetation coverage effects. To do this, satellite data needs to be presented in a convenient form with a number of pre-processing operations such as selection of: the particular channel; area of interest; and time and date of interest (or a sequence of times and dates). It is also important that once specified, such snapshots or sequences can be easily fed into applications programs which can carry out further processing to derive composite imagery or some item of metadata. We illustrate this idea using an example of computing an approximate percentage cloud cover figure for a selected geographic area and time and date. We describe our repository of geostationary satellite data in section B.2 and some of the operations we wish users to be able to carry out in section B.3.

We have implemented our infrastructure using the Common Gateway Interface

(CGI) [100] to a standard WWW server such as the NCSA **httpd** daemon run on a UNIX platform and also Java applets which can be down-loaded and run on the WWW client browser. This architecture gives a good compromise between a system that can be rapidly prototyped using existing server-side utilities and the desired client-side functionality that we require.

A number of distributed and parallel processing technologies can be embedded in our infrastructure. We use remote execution in the form of distributed **rsh** invocations to farm out parts of the processing to other workstations in our cluster. Parallel programs running as remote services either on the workstation farm or on a dedicated machine are also discussed. We describe the infrastructure architecture in section B.4. We have made some measurements of the performance of various parts of our system and describe these in section B.5 along with the tradeoffs that arise.

There are two important data access rates that limit our prototype system. Primarily, the system is limited by data transfer rates between the component machines that are used to provide the processing services. Our implemented system links these machines together using shared file systems and Asynchronous Transfer Mode (ATM) communications [103] running at 155Mbps. The second access speed limitation arises from the available bandwidth between the WWW client and server. In the case of down-loading large images or movie sequences this can be significant, but for information services such as browsing catalogues of small sub-sampled imagery and metadata, this bandwidth is not a significant restriction. We have also experimented with accessing the system over wide area networks such as over Telstra's Experimental Broadband Network [109, 110, 140].

In section B.6 we discuss the general abstract model for an infrastructure like Eric and how we plan to adapt it and extend it to provide more general services. We also summarise our findings and conclusions.

B.2 Repository Design

In this section we describe our repository system for organising the storage of satellite imagery from the Japanese GMS5 satellite. We currently obtain this data by mirroring a NASA ftp site and we use a series of Unix shell scripts to maintain separate files for each time and date and channel of data. Files are stored in the Hierarchical Data Format (HDF) developed by NCSA [165].

The HDF file format is supported by a set of utilities and programming libraries which we have integrated into our Eric infrastructure.

The Geostationary Meteorological Satellite (GMS) provides more than 24 full hemisphere multichannel images per day, requiring approximately 204MBytes storage capacity per day, or 75GBytes per year. The GMS-5 provides visual and infra-red data.

More information on the repository is given in [127] and the GMS satellite and its sensors are described in the Japanese Meteorological Agency (NASDA) User Manual for the satellite [133].

At present, the Eric system accesses non pre-processed GMS5 data that is stored in a flat directory structure on a UNIX file-system, where each file name reflects its data channel and time-date-stamp. Each HDF file contains the image and metadata sufficient to identify the satellite orbit and attitude configuration and thus allow geo-rectification or conversion from image pixel coordinates to latitude and longitude coordinates. We plan a future capability that will either access a secondary repository of latitude/longitude geo-rectified data, or will carry out the conversion on demand [129].

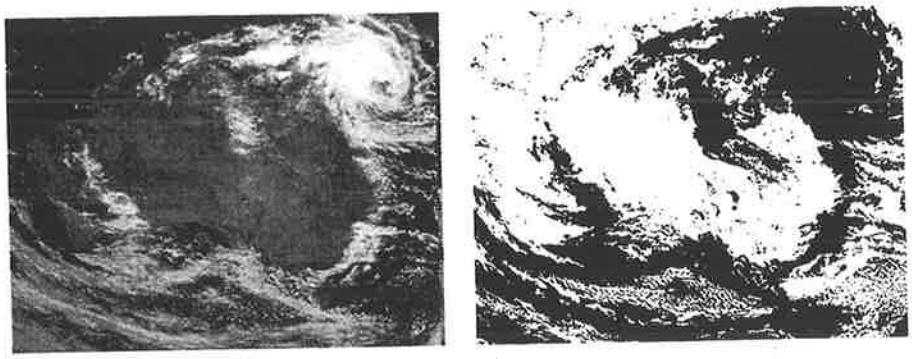
Our present repository system involves storage of the most recent image data (approximately one half-year of data) on a 100Gigabyte StorageWorks RAID, with older data archived on tapes controlled automatically by a 1.2Terabyte StorageTek tape silo. We are currently developing an archive control system that will migrate data automatically to tapes both locally and to an additional remote tape silo accessible through the Experimental Broadband Network.

We envisage that a repository system such as we describe may have additional uses when coupled to other information sources such as weather services. Integrated systems making use for numerical weather prediction applications may assimilate satellite imagery as part of their operation and produce derived data products such as forecasts or flow fields [104].

B.3 Imagery Operations

The original driving reason for developing our system was to allow simple browsing of the satellite imagery, choosing particular images of scientific interest. The simple queries supported under the original system were to be requests for a particular image channel (such as visual, infra-red1, infra-red2 or water-vapour) at a particular time

and date. The full images returned were however of the whole hemisphere of the Earth visible from the satellite. More useful for scientific purposes is to request a particular image resolution and sub-area of interest, as shown in figure 57(i).



i) ii)

Figure 57: i) GMS-5 Visible Spectra View of Australia, and ii) corresponding cloud bit mask

For visual inspection (or browsing) of snapshot images, it is useful to create 'thumbnail images' of reduced resolution - effectively sub-sampled images. Our present system creates these dynamically from the primary data images, caching results from previous queries. We are investigating the tradeoffs from creating a complete secondary image repository of a standard thumbnail image size and thus saving the processing time to recreate thumbnails dynamically.

It is also useful for scientific analysis of the data sets to specify a query involving a sequence of images. Such a query is specified by a start and end time and date as well as a stride. The query may then be for "every image at midnight, for the last week".

For browsing a sequence of images it is useful to create a series of thumbnail resolution images that can be played through the WWW interface as a movie sequence. We employ the MPEG file format for this and integrate public domain MPEG-creation utilities into our infrastructure.

Alternatively, the system can return what is essentially a vector of static images that can be processed by some application program using the same user-specified parameters. This is a convenient mechanism to specify a batch process once a processing algorithm has been specified by the user. This mechanism allows composite images to be made into a movie sequence as well as the raw imagery.

Reduction operations can also be incorporated into the infrastructure. A reduction operation is one which reduces a two dimensional data item such as an image into a vector or scalar quantity. An example would be an algorithm to evaluate percentage cloud cover. We have implemented this very simplistically as a computational example. A simple definition of cloud in the image data is given by 'cold, bright pixel vales'. A simplistic algorithm would determine whether a pixel shows cloud or not by thresholding the visual channel for brightness value and the thermal infra-red channel for a temperature value, as shown in figure 57(ii).

We are developing an operator language that allows a user of the system (or an application programmer) to specify the sequence of image transforms and reduction operations to be applied to a single image or sequence of images from the repository. Since these operations can be implemented as high performance computing modules we believe our infrastructure provides a useful testbed for investigating parallel and distributed algorithms for these applications.

B.4 Eric Architecture

In this section we discuss the overall architecture of our Eric system and some of the important issues we have identified for developing an improved system. Eric was originally implemented as a single driver program running on the WWW server machine and invoked as a CGI [100] program by the httpd web server daemon. The driver script was a Perl program, and some of the necessary functions were implemented as C programs or shell scripts embedded as system calls from the Perl [232] script. This mechanism allows for very rapid prototyping but has a number of disadvantages. Perl is not a particularly good language for maintaining large software projects. Furthermore embedded system calls are not very efficient either in memory nor in computational startup costs, since they must employ a separately spawned UNIX process to handle them.

Our improved architecture (see figure B.4(i)) made use of a combined Web server and Java application server to work together to service user requests using WWW protocols. The WWW server and Java application both share a common file space and can therefore communicate using shared information contained within a file.

The Eric system provides users with an HTML form that can be used to request particular information products from the database of satellite imagery. A sequence of forms is used to narrow down the particular options relevant to a hierarchy of choices

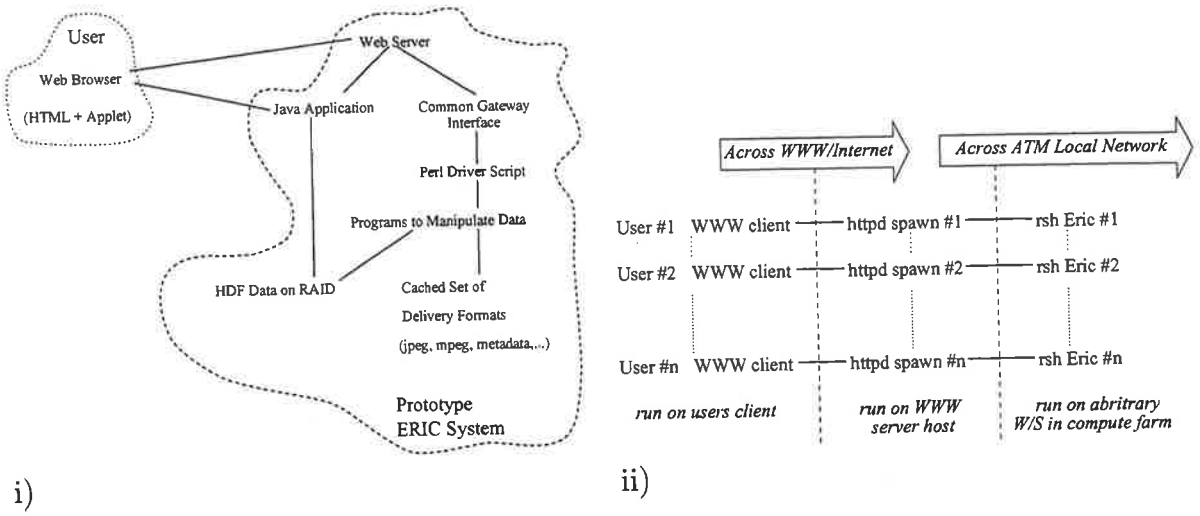


Figure 58: i)Eric: Combined CGI and Java Infrastructure , ii)Multiple Server daemon processes spawned to handle multiple users

presented to the user. At each stage, the HTML form presented to the user has links to help information and to the main menu of Eric options. Data is delivered back to the user directly as an embedded image or movie or text file. Once these have been transferred to the user's browser, he can choose to store them to his own local file system for future use in addition to them being automatically stored in the WWW browser cache and Eric's product cache. Some of the ERIC interfaces, for selecting and displaying a single band satellite image, are shown in figures 59 and 60.

In the case of popular products and for a system that may be serving a large community of users, there is a potential performance improvement in managing caches of final and intermediate products. Our system does this by using a data oriented naming scheme that can be parsed at any stage by the driver script. For computationally intensive operations, this makes it possible to determine whether the result has already been computed and is in the cache. Our present system uses a simple naming scheme based on adding many details to the filenames used - a more robust server based database approach would be preferable since it would allow better organisation of the cached data than a flat file structure.

The present Eric system makes heavy use of temporary files to handle multiple user requests and to track the intermediate results between programs chained together to provide the various services Eric offers. This requires use of process identifiers and other temporary tokens to track ownership and description of these files. This

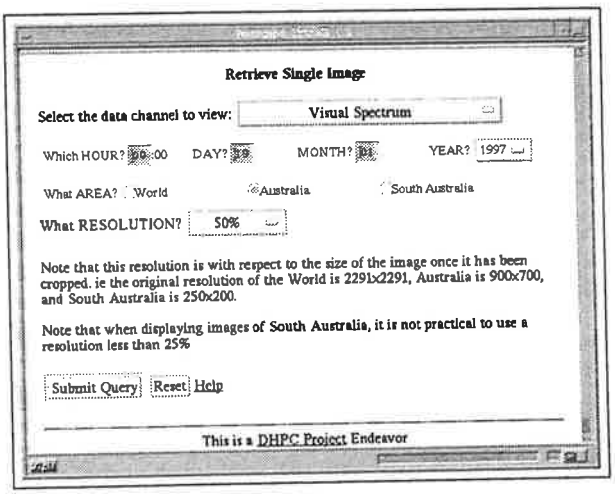


Figure 59: ERIC: Single image query form.

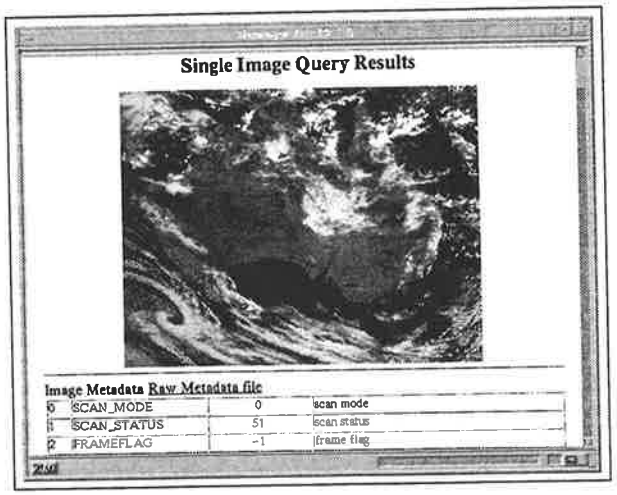


Figure 60: ERIC: Single image results screen.

approach does not scale well. We implement a primitive form of garbage collection by assigning the UNIX **cron** daemon the task of periodically clearing out outdated temporary information from Eric cache areas as well as older information if the cache area is becoming full.

Error handling in CGI programming in general and Eric in particular is not robust. Most WWW servers maintain a log of error codes should the CGI program fail to run at all, but a great deal of checking for file existence and explicit coding of error reporting is necessary to keep the user informed should a complex request fail. Reporting the exact conditions is possible and most failures in our present system occur due to a missing image in the master database of GMS data.

We describe some of the implementation techniques for Eric in [127,129]. The Eric

system presents users with a familiar HTML forms interface. Currently, the service requests our system can handle are coded explicitly in the cgi-bin driver script. We are developing the idea of generalised service specification language that can be used to generate the sequence of interface forms to specify the request automatically. This will allow easier creation and maintenance of a static list of services as well as the interesting possibility of allowing dynamic addition of services to a running system.

B.5 Performance Issues

In this section we present some timing analysis of the various functions our system carries out. The (wall-clock) times listed are measured in seconds and measurements were made using the Unix **time** utility. Table 11 illustrates the time taken to uncompress a raw imagery file from the repository and convert it to a user transmission format such as JPEG for various image resolutions and sub areas of interest. These times are acceptable for delivery of single images in interactive time, but not for sequences of images.

Magnification	In-Cache Convert Time			Out-of-Cache Convert Time		
	SA	AUS	WORLD	SA	AUS	WORLD
0.25	1.7±.2	1.7±.2	1.6±.2	20±1	32±7	32±6
1.00	1.9±.2	1.7±.1	1.9±.2	19±3	26±1	29±1
2.00	2.3±1	1.8±.2	1.8±.2	17±2	35±8	77±27

Table 11: Single image compressed HDF to JPEG conversion times for images

Some program operations such as encoding a sequence of static images into an MPEG movie are limited by the capabilities of the processing host. This can be addressed by implementing a parallel or distributed MPEG encoder or identifying a more powerful encoder host that can provide this service. MPEG performance is discussed in [44].

An important issue for many of the operations provided by the Eric system is the time taken to load data into programs from disk. Our primary data repository is presently required to store compressed data. This adds an additional decompression overhead onto processing operations. The cloud cover operation is particularly expensive since it requires *two* image channels.

	SA	AUS	WORLD
In Cache	3.7±1.5	3.0±.6	2.9±.9
Not In Cache	258±110	286±85	396±51

Table 12: Times to compute/retrieve cloud cover for Australia

The times in table 12 show the computational times for calculating cloud cover for Australia. The times are very high compared with those for accessing previously cached result. The compute times can be reduced significantly by use of the **rsh** and parallel processing techniques described in [129]. However, even a parallel application is limited by the time it takes to load the data, on-demand from the repository.

B.6 Discussion and Conclusions

Our main objective in constructing the Eric system was as a discussion vehicle with potential users of satellite imagery and other large on-line data archive systems [108]. A number of issues have emerged from this work and have stimulated our plans for other prototype demonstrator systems [114].

The functionality of the Eric system is sufficient to deliver a range of simple data products to a remote user using relatively simple product specification forms. We believe a more general system would need to make use of an intermediate language for product specification rather than requiring all capabilities to be hard coded into the delivery mechanism. User feedback indicates that an important advantage of the Eric system is in hiding the computational details required to meet a high-level user request.

Implementing the Eric system has identified a number of design and software engineering issues that we expect will aid our next prototype development. Parallelism and distributed computation were added to Eric after the original design. This raised a number of difficulties in maintaining the integrity of temporary or cached results amongst different hosts sharing the file system and amongst different processes servicing separate user requests. We anticipate that use of a cleaner client/server based model for the functional components of the system will alleviate this problem.

The issue of placement of Eric processes was quite difficult. Distribution was achieved through statically configuring a remote processor to provide some of the computation necessary to fill a query. The `mpeg_encode` program has a facility for

distributing the processing of frames across processors. Unfortunately, the processors must be nominated in a parameter file, which the program processes. This early technology prevents failure recovery when hosts are unavailable or become unavailable during the computation. Another limiting factor was that the both Eric and the mpeg encoding program relied on the use of shared file systems.

In conclusion, we believe our prototype system has exceeded our expectations as a discussion vehicle and has allowed us to work with colleagues wishing to use the GMS5 imagery in applied science areas. This application of large data objects such as GMS5 images has also provided a fruitful driver for our investigation of design issues for a more general data delivery system.

Bibliography

[1] D. Abramson, R. Sosic, J. Giddy, and M. Cope. The Laboratory Bench: Distributed Computing for Parameterised Simulations. In *Proc. 1994 Parallel Computing and Transputers Conf*, pages 17–27, November 1994.

[2] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A Tool for Performing Parameterised Simulations using Distributed Workstations. In *Proc. 4th IEEE Symp. High Performance Distributed Computing*, Virginia, August 1995.

[3] David Abramson and Jon Giddy. Scheduling Large Parameteric Modelling Experiments on a Distributed Meta-computer. In *Proc. PCW '97*, September 1997.

[4] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A Comparison of List Schedules for Parallel Processing Systems. *Communications of the ACM*, 17(12):685–690, December 1974.

[5] D. A. Adams. A computational model with dataflow sequencing. Technical Report Technical Report CS 117, Computer Science Department, Stanford University, 1968.

[6] Ishfaq Ahmad. Editorial: Resource Management in Parallel and Distributed Systems with Dynamic Scheduling: Dynamic Scheduling. *Concurrency: Practice and Experience*, 7(7):587–590, October 1995.

[7] Ishfaq Ahmad. Editorial: Resource Management of Parallel and Distributed Systems with Static Scheduling: Challenges, Solutions and New Problems. *Concurrency: Practice and Experience*, 7(5):339–347, August 1995.

[8] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, 1986. ISBN 0-201-10194-7.

[9] Paul Albitz and Cricket Liu. *DNS and BIND.* O'Reilly and Associates, Inc., 1992. ISBN 1-56592-010-4.

[10] Yair Amir, Baruch Awerbuch, and Ryan S. Borgstrom. A Cost-Benefit Framework for Online Management of a Metacomputing System. In *Proc. 1st Int. Conf. Information and Computation Economies (ICE-98)*, October 1998.

[11] Yair Amir, Baruch Awerbuch, and Ryan S. Borgstrom. The Java Market: Transforming the Internet into a Metacomputer. Technical Report CNDS-98-1, Johns Hopkins University, 1998.

[12] Thomas E. Anderson, David E. Culler, and the NOW Team. A Case for NOW(Networks of Workstations). *IEEE Micro*, December 1994.

[13] ANU Supercomputer Facility. Fujitsu VPP300 Job Turn Around Statistics. WWW Page, Last visited 27 May 1999. Available from http://anusf.anu.edu.-au/VPP/queuestats/.

[14] ANU Supercomputer Facility. SGI Power Challenge Job Turn Around Statistics. WWW Page, Last visited 27 May 1999. Available from http://-anusf.anu.edu.au/PC/queuestats/.

[15] Ken Arnold, Bryan O'Sullivan, Robert W. Schiefler, Jim Waldo, and Ann Wollrath. *The Jini Specification.* The Jini Technology Series. Addison Wesley Longman, June 1999. ISBN 0-201-61634-3.

[16] Andrea C. Arpaci-Dusseau, David E. Culler, and Alan Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *1998 SIGMETRICS Conf. the Measurement and Modeling of Computer Systems*, pages 233–243, June 1988.

[17] Australian Centre for Remote Sensing. ACRES Digital Catalogue. Available from http://acs.auslig.gov.au/intro.html, last visited March 1999.

[18] Advanced Visual Systems (AVS). *AVS Developer's Guide.* Advanced Visual Systems Inc, release 4 edition, May 1992. Part Number: 320-0013-02, Rev B.

[19] Mark A. Baker and Geoffery C. Fox. *Metacomputing: Harnessing Informal Supercomputers.* High Performance Cluster Computing. Prentice-Hall, May 1999. ISBN 0-13-013784-7.

[20] Mark A. Baker and Geoffrey C. Fox. Distributed Cluster Computing Environments. Available from http://www.npac.syr.edu/~mab/homepage/-cluster-computing/, last visited January 1996.

[21] Mark A. Baker, Geoffrey C. Fox, and Hon W. Yau. A Review of Commercial and Research Cluster Management Software. Technical Report SCCS-0748, Northeast Parallel Architectures Center, June 1996.

[22] B. Bauer and F. Ramme. A general purpose Resource Description Language. In R. Grebe and M. Baumann, editors, *Proc. TAT'91*, pages 68–75. Springer Verlag, 1991.

[23] A. Bayucan, R. L. Henderson, T. Proett, D. Tweten, and B. Kelly. Portable Batch System External Reference Specification. NAS Scientific Computing Branch, NASA Ames Research Center, California, June 1996.

[24] Fran Berman, Rich Wolski, Silva Figueira, Jennifer Schopf, and Gary Shao. Application-Level Scheduling on Distributed Heterogeneous Networks. In *Supercomputing '96*, November 1996.

[25] Fran Berman and Rick Wolski. The AppLeS Project: A Status Report. Available from http://www-cse.ucsd.edu/groups/hpcl/apples/hetpubs.html, 1998.

[26] Dimple Bhatia, Vanco Burzevski, Maja Camuseva, Geoffrey Fox, Wojtek Fumanski, and Girish Premchandran. WebFlow - A Visual Programming Paradigm for Web/Java Based Coarse Grain Distributed Computing. In *Proc. Workshop on Java for Computational Science and Engineering*, December 1996.

[27] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Trans. Computer Systems*, 2(1):39–59, February 1984.

[28] S. H. Bokhari. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE Trans. Software Engineering*, SE-7(6):583–589, November 1981.

[29] Shahid H. Bokhari. On the Mapping Problem. *IEEE Trans. Computers*, C-30(3):207–214, March 1981.

[30] Shahid H. Bokhari. Partitioning Problems in Parallel, Pipelined and Distributed Computing. *IEEE Trans. Computers*, 37(1):48–57, January 1988.

[31] Matthias Brune, Jörn Gehring, and Alexander Reinefeld. Heterogeneous Message Passing and a Link to Resource Management. *J. Supercomputing*, 11:1–17, 1997.

[32] Ralph M. Butler and Ewing L. Lusk. Monitors, Messages, and Clusters: the p4 Parallel Programming System. Available from http://www-fp.mcs.anl.gov/-~lusk/p4/p4-paper/paper.html, 1991.

[33] Denis Caromel. Toward a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[34] T. L. Casavant and J. G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems, May 1986.

[35] Centre National d'Etudes Spatiales. SPOT satellite Earth Observation System. Available from http://sads.cnes.fr/ceos/cdrom-98/ceos1/satellit/-spotsys/english/s_syst.htm, last visited March 1999.

[36] K. Mani Chandy, Anand Chelian, Boris Dimitrov, Zuzana Dobes, John Garnett, Joseph Kiniry, Huy Le, Jacob Mandelson, Matthew Richardson, Adam Rifkin, Eve Schooler, Paolo A.G. Sivilotti, Wesley Tanaka, and Luke Weisman. A New Approach To Collaborative Distributed Computing. *CRPC Newsletter*, 1996.

[37] K. Mani Chandy, Joseph Kiniry, Adam Rifkin, and Daniel Zimmerman. Webs of Archived Distributed Computations for Asynchronous Collaboration. *J. Supercomputing*, 11(2):101–118, 1997.

[38] K. Mani Chandy and Adam Rifkin. Systematic Composition of Objects in Distributed Internet Applications: Processes and Sessions. In *Proc. HICSS 30*, 1997.

[39] K. Mani Chandy, Adam Rifkin, Paolo A.G. Sivilotti, Jacob Mandelson, Matthew Richardson, Wesley Tanaka, and Luke Weisman. A World-Wide Distributed System Using Java and the Internet. In *High Performance Distributed Computing (HPDC-5) 1996*. Caltech, March 1996.

[40] Song Chen and Mary M. Eshaghian. A fast recursive mapping algorithm. *Concurrency: Practice and Experience*, 7(5):391–409, August 1995.

[41] Timothy C. K. Chou and Jacob A. Abraham. Load Balancing in Distributed Systems. *IEEE Trans. Software Engineering*, SE-8(4):401–412, July 1982.

[42] Yuan-Chieh Chow and Walter H. Kohler. Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System. *IEEE Trans. Computers*, C-28(5):354–361, May 1979.

[43] P. D. Coddington, K. A. Hawick, and H. A. James. Web-Based Access to Distributed, High-Performance Geographic Information Systems for Decision Support. Technical Report DHPC-037, Advanced Computational Systems CRC, Department of Computer Science, University of Adelaide, June 1998.

[44] P. D. Coddington, K. A. Hawick, H. A. James, and F. A. Vaughan. Movie Sequences of Archived Imagery on Distributed, High-Performance Computing Systems. Technical Report DHPC-012, Computer Science Department, University of Adelaide, August 1997.

[45] P. D. Coddington, K. A. Hawick, K. E. Kerry, J. A. Mathew, A. J. Silis, D. L. Webb, P. J. Whitbread, C. G. Irving, M. W. Grigg, R. Jana, and K. Tang. Implementation of a Geospatial Imagery Digital Library using Java and CORBA. In *Proc. Technologies of Object-Oriented Languages and Systems Asia (TOOLS 27)*. IEEE, September 1998.

[46] David E. Culler. Castle Project. Available from http://http.cs.berkeley.edu/-projects/parallel/castle/, last visited 8 June 1999.

[47] Zarka Cvetanovic. The Effects of Problem Partitioning, Allocation, and Granularity on the Performance of Multiple-Processor Systems. *IEEE Trans. Computers*, C-36(4):421–432, April 1987.

[48] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A Resource Management Architecture for Metacomputing Systems. In Dror G. Feitelson and Larry Rudolph, editors, *JSSPP'98*, number 1459 in LNCS, pages 62–82. Springer-Verlag Berlin Heidelberg, 1998.

[49] H. G. Dietz, W. E. Cohen, and B. K. Grant. Would you run it here... or there? (AHS: Automatic Heterogeneous Supercomputing). In *Int. Conf. Parallel Processing, Vol II: Software*, pages 217–221, 1993.

[50] Digital Equipment Corporation, Intel Corporation, Xerox Corporation. The Ethernet - A Local Area Network, Version 1.0, September 1980.

[51] Dublin Core Working Group. Dublin Core Metadata. Available from http://purl.org/metadata/dublin_core/, last visited March 1999.

[52] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective Distributed Scheduling of Parallel Workloads. *Proc. SIGMETRICS '96*, 1996.

[53] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Trans. Software Engineering*, SE-12(5):622–675, May 1986.

[54] Hesham El-Rewini. *Parallel and Distributed Computing Handbook*, chapter Partitioning and Scheduling, pages 239–273. Computer Engineering Series. McGraw-Hill, 1996. ISBN 0-07-073020-2.

[55] Hesham El-Rewini and Hesham H. Ali. Static Scheduling of Conditional Branches in Parallel Programs. *J. Parallel and Distributed Computing*, 24:41–54, 1995.

[56] Hesham El-Rewini, Hesham H. Ali, and Ted Lewis. Task Scheduling in Multiprocessing Systems. *IEEE Computer*, pages 27–37, December 1995.

[57] Hesham El-Rewini and T. G. Lewis. Scheduling Parallel Program Tasks onto Arbitrary Machines. *J. Parallel and Distributed Computing*, 9:138–153, 1990.

[58] Robert Englander. *Developing Java Beans*. O'Reilly, 1997. ISBN 1-56592-289-1.

[59] ESRI. Arc/Info. Available from http://www.esri.com/software/arcinfo/-index.html, last visited April 1999.

[60] European Space Agency. Earthnet Online, ESA Multimission Remote Sensing Product Catalogue. Available from http://gds.esrin.esa.it/reg/catalogue.html, last visited March 1999.

[61] Graham E Fagg, Keith Moore, Jack J. Dongarra, and Al Geist. Scalable Networked Information Processing Environment (SNIPE). In *Proc. SuperComputing 97*, 1997.

[62] Dror G. Feitelson. Job Scheduling in Multiprogrammed Parallel Systems. Technical Report IBM Research Report RC 19790 (87657), IBM T.J. Watson Research Center, Yorktown Heights, NY, October 1994. Extended Version.

[63] Dror G. Feitelson and Larry Rudolph. Parallel Job Scheduling: Issues and Approaches. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *LNCS*, pages 1–18. Springer-Verlag, 1995.

[64] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *LNCS*, pages 1–34. Springer-Verlag, 1997.

[65] David Fernández-Baca. Allocating Modules to Processors in a Distributed System. *IEEE Trans. Software Engineering*, 15(11):1427–1436, November 1989.

[66] Fritz Ferstl. Industrial experiences with Codine. In *Proc. Workshop on Distributed Computing*. Amsterdam Science Park, January 1997. Available from http://www.wins.uva.nl/research/DC97/ferstl/.

[67] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith, and Steven Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In IEEE, editor, *Proc. 6th IEEE Symp. High-Performance Distributed Computing 1997*, 1997.

[68] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. Available from http://www.mpi-forum.org.

[69] Ian Foster and Carl Kesselman. Globus: A Meta-computing Infra-structure Toolkit. *Int. J. Supercomputer Applications*, 1996.

[70] Ian Foster and Carl Kesselman. The Globus Project: A Status Report. In *Proc. Heterogeneous Computing Workshop*, pages 4–18. IEEE Computer Society Press, 1998.

[71] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1999. ISBN 1-55860-475-8.

[72] Ian Foster, Carl Kesselman, Craig Lee, Bob Lindell, Klara Nahrstedt, and Alain Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. Available from http://www.globus.org.

[73] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Parallel and Distributed Computing*, 1996.

[74] Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina. *Parallel Computing Works!* Morgan Kaufmann Publishers, Inc., 1994. ISBN 1-55860-253-4.

[75] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice.* The Jini Technology Series. Addison Wesley Longman, June 1999. ISBN 0-201-30955-6.

[76] D. K. Friesen and M. A. Langston. Analysis of a Compound Bin Packing Algorithm. *SIAM Journal on Discrete Mathematics*, 4:61–79, 1991.

[77] Cong Fu, Tao Yang, and Apostolos Gerasoulis. Integrating Software Pipelining and Graph Scheduling for Iterative Scientific Computations. In *Proc. Irregular '95*, number 980 in LNCS, pages 127–141, September 1995.

[78] Jörn Gehring and Alexander Reinefeld. MARS - A Framework for Minimizing Job Execution Time in a Metacomputing Enviroment. *Future Generation Computer Systems (FGCS)*, 12(1):87–99, 1996.

[79] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine A Users' guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[80] David Gelernter. Parallel Programming in Linda. Technical Report 359, Yale University Department of Computer Science, January 1985.

[81] Genias Software GmbH. CODINE: Computing in Distributed Networked Environments. Available from http://www.genias.de/products/codine/-description.html, last visited July 1999.

[82] Genias Software GmbH. Global Resource Director. Available from http://-www.genias.de/products/grd/grd_index.html, last visited July 1999.

[83] Genias Software GmbH. PerfStat. Available from http://www.genias.de/-products/perfstat/ps_index.html, last visited July 1999.

[84] Apostolos Gerasoulis, Jia Jiao, and Tao Yang. A multistage approach to scheduling task graphs. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 22:81–103, 1995.

[85] Apostolos Gerasoulis, Jia Jiao, and Tao Yang. Scheduling of structured and unstructured computation. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 21:139–172, 1995.

[86] Apostolos Gerasoulis and Tao Yang. A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors. *J. Parallel and Distributed Computing*, 16:276–291, 1992.

[87] Kevin L. Gong and Lawrence A. Rowe. Parallel MPEG-1 Video Encoding. In *Proc. PCS '94*, 1994.

[88] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison Wesley Longman, 1996. ISBN 0-201-63451-1.

[89] Paul A. Gray and Vaidy S. Sunderam. IceT: Distributed Computing and Java. *Concurrency, Practice and Experience*, 9(11), November 1997.

[90] Paul A. Gray and Vaidy S. Sunderam. Developing Technologies for Broad-Network Concurrent Computing. *J. Systems Architecture*, to appear.

[91] Thomas P. Green. DQS User Interface Preliminary Design Document. Supercomputer Computations Research Institute, Florida State University, July 1993.

[92] Andrew S. Grimshaw. An Introduction to Parallel Object-Oriented Programming with Mentat. Technical Report CS-91-07, University of Virginia, April 1991.

[93] Andrew S. Grimshaw, Adam Ferrari, and Emily West. *Parallel Programming Using C++*, chapter Mentat, pages 383–427. The MIT Press, Cambridge, Massachusetts, 1996.

[94] Andrew S. Grimshaw and Wm. A. Wulf and the Legion team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1), January 1997.

[95] Andrew S. Grimshaw and Wm. A. Wulf. Legion – A View From 50,000 Feet. In *Proc. Fifth IEEE Int. Symp. High Performance Distributed Computing*, Los Alamos, California, August 1996. IEEE Computer Society Press.

[96] Andrew S. Grimshaw, Wm. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds. Legion: The Next Logical Step Toward a Nationwide Virtual Computer. Technical Report CS-94-21, Computer Science Department, University of Virginia, June 1994.

[97] W. Gropp, E. Lusk, N. Doss, and A. Sjkellum. *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard.* Argonne National Laboratories, 1996.

[98] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1994. ISBN 0-262-57104-8.

[99] Duncan A. Grove, Andrew J. Silis, J.A. Mathew, and K.A. Hawick. Secure Transmission of Portable Code Objects in a Metacomputing Environment. In *Proc. Int. Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, July 1998.

[100] Shishir Gundavaram. *CGI Scripting on the World Wide Web.* O'Reilly & Associates, 1996. ISBN 1-56592-168-2.

[101] Babak Hamidzadeh and David J. Lilja. Self-adjusting scheduling: an on-line optimization technique for locality management and load-balancing, Vol II: Software. In *Proc. Int. Conf. Parallel Processing*, pages 39–46, 1994.

[102] Babak Hamidzadeh, David J. Lilja, and Yacine Atif. Dynamic scheduling techniques for heterogeneous computing systems. *Concurrency: Practice and Experience*, 7(7):633–652, October 1995.

[103] R. Handel, M. N. Huber, and S. Schroder. *ATM Networks - Concepts, Protocols, Applications.* Addison Wesley Longman, 1994. ISBN 0-201-42274-3.

[104] K. A. Hawick, R. S. Bell, A. Dickinson, P. D. Surry, and B. J. N. Wylie. Parallelisation of the Unified Model Data Assimilation Scheme. In *Proc. Workshop of Fifth ECMWF Workshop on Use of Parallel Processors in*

Meteorology, Reading, November 1992. European Centre for Medium Range Weather Forecasting (ECMWF).

[105] K. A. Hawick and P. D. Coddington. High-Performance Fortran Libraries and Services for Numerical Simulations in Computational Science. Technical Report DHPC-036, Advanced Computational Systems CRC, Department of Computer Science, University of Adelaide, March 1998.

[106] K. A. Hawick and H. A. James. Distributed High-Performance Computation for Remote Sensing. In *Proc. Supercomputing 97*, November 1997.

[107] K. A. Hawick and H. A. James. A Distributed Job Placement Language. Technical Report DHPC-070, Department of Computer Science, The University of Adelaide, May 1999.

[108] K. A. Hawick, H. A. James, K. J. Maciunas, F. A. Vaughan, A. L. Wendelborn, M. Buchhorn, M. Rezny, S. R. Taylor, and M. D. Wilson. Geographic Information Systems Applications on an ATM-Based Distributed High Performance Computing System. In HPCN, editor, *Proceedings HPCN'97*, Vienna, Austria, August 1997.

[109] K. A. Hawick, H. A. James, K. J. Maciunas, F. A. Vaughan, A. L. Wendelborn, M. Buchhorn, M. Rezny, S. R. Taylor, and M. D. Wilson. An ATM-based Distributed High Performance Computing System. In HPCN, editor, *Proceedings HPCN'97*, Vienna, Austria, August 1997. IEEE Computer Society Press.

[110] K. A. Hawick, H. A. James, K. J. Maciunas, F. A. Vaughan, A. L. Wendelborn, M. Buchhorn, M. Rezny, S. R. Taylor, and M. D. Wilson. Geostationary-Satellite Imagery Applications on Distributed, High-Performance Computing. In *Proc. HPCAsia'97*, August 1997.

[111] K. A. Hawick, H. A. James, and J. A. Mathew. Remote Data Access in Distributed Object-Oriented Middleware. *To appear in Parallel and Distributed Computing Practices*, 1999.

[112] K. A. Hawick, H. A. James, C. J. Patten, P. D. Coddington, and A. J. Silis. Middleware and Research Issues for Web-based Computing. Technical Report DHPC-067, Department of Computer Science, The University of Adelaide, May 1999.

[113] K. A. Hawick, H. A. James, A. J. Silis, D. A. Grove, K. E. Kerry, J. A. Mathew, P. D. Coddington, C. J. Patten, J. F. Hercus, and F. A. Vaughan. DISCWorld: An Environment for Service-Based Metacomputing. *Future Generation Computing Systems (FGCS)*, 15:623–635, 1999.

[114] K. A. Hawick and F. A. Vaughan. DISCWorld - Distributed Information Systems Cloud of High Performance Computing Resources - Concepts Discussion Document. Technical report, Advanced Computational Systems CRC, Department of Computer Science, University of Adelaide, December 1996. DHPC Technical Report.

[115] K.A. Hawick, D.A. Grove, and F.A. Vaughan. Beowulf — A New Hope for Parallel Computing? Technical Report DHPC-061, Advanced Computational Systems Cooperative Research Center, Department of Computer Science, University of Adelaide, January 1999.

[116] R. L. Henderson and D. Tweten. Portable Batch System Requirements Specification. NAS Scientific Computing Branch, NASA Ames Research Center, California, April 1995.

[117] S. Herbert. Official Administrator's Guide to Generic NQS. Sterling Software, September 1994.

[118] P. Homburg, M. van Steen, and A.S. Tanenbaum. An Architecture for a Wide Area Distributed System. In *Proc. Seventh ACM SIGOPS European Workshop*, pages 75–82, September 1996.

[119] A. R. Hurson, B. Lee, B. Shirazi, and M. Wang. A Program Allocation Scheme for Data Flow Computers. *Proc. 1990 Int. Conf. Parallel Processing*, I:I–415–I–423, 1990.

[120] IEEE CS Task Force on Cluster Computing. TFCC Newsletter Dialogs on TFCC-L, Vol 1, No 1. Available from http://www.eg.bucknell.edu/~hyde/-tfcc/vol1no1-dialog.html, April 1999.

[121] Informix Corporation. Web Page. Available from http://www.informix.com, last visited April 1999.

[122] International Business Machines. Distributed Metadata Management in the High Performance Storage System. Available from http://www.clearlake.ibm.-com/houston/metadata.html, Last visited May 1999.

[123] International Business Machines Corporation. LoadLeveler. Network job scheduling and job management.Available from http://www.austin.ibm.com/-software/sp_products/loadlev.html, Last visited May 1998.

[124] International Business Machines Corporation. XML Parser for Java v1.1.14. Available from http://www.alphaWorks.ibm.com/formula/xml, February 1999.

[125] ISO. ISO Technical Committee 211 on Geographic Information/Geomatics. Available from http://www.statkart.no/isotc211/, last visited March 1999.

[126] R. Jagannathan. *Parallel and Distributed Computing Handbook*, chapter Dataflow Models, pages 223–238. Computer Engineering Series. McGraw-Hill, 1996. ISBN 0-07-073020-2.

[127] H. A. James and K. A. Hawick. Eric: A User and Applications Interface to a Distributed Satellite Data Repository. Technical Report DHPC-008, Department of Computer Science, The University of Adelaide, April 1997.

[128] H. A. James and K. A. Hawick. Resource Descriptions for Job Scheduling in DISCWorld. In *Proc. Integrated Data Environments Australia (IDEA5) Workshop*, 1998.

[129] H. A. James and K. A. Hawick. A Web-based Interface for On-Demand Processing of Satellite Imagery Archives. In Chris McDonald, editor, *Proc. 21st Australasian Computer Science Conf. ACSC'98*, volume 20 of *Australian Computer Science Communications*. Springer-Verlag Pte Ltd, February 1998.

[130] H. A. James and K. A. Hawick. Remote Application Scheduling on Metacomputing Systems. Technical Report DHPC-064, Department of Computer Science, University of Adelaide, Department of Computer Science, University of Adelaide, February 1999.

[131] H. A. James, K. A. Hawick, and P. D. Coddington. Scheduling Independent Tasks on Metacomputers. In S. Olariu and J. Wu, editors, *Proc. ISCA 12th Int. Conf. on Parallel and Distributed Computing Systems*, pages 156–162. The

International Society for Computers and Their Applications - ISCA, August 1999. ISBN 1-880843-29-3.

[132] Heath A. James. Using ATM In Distributed Applications. Honours Thesis, Department of Computer Science, The University of Adelaide, 1995.

[133] Japanese Meteorological Satellite Center. The GMS User's Guide. 3-235 Nakakiyoto, Kiyose, Tokyo 204, Japan, 1989.

[134] Didier Le Gall Joan L. Mitchell and Chad Fogg. *MPEG Video Compression Standard.* Chapman & Hall, 1996.

[135] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. Information Processing 74*, pages 471–475, 1974.

[136] Gilles Kahn and David B. MacQueen. Coroutines and Networks of Parallel Processes. In B. Gilchrist, editor, *Proc. Information Processing 77*, pages 993–998, 1977.

[137] John F. Karpovich. Support for Object Placement in Wide Area Heterogeneous Distributed Systems. Technical Report Technical Report CS-96-03, University of Virginia Department of Computer Science, January 1996.

[138] Marc T. Kaufman. An Almost-Optimal Algorithm for the Assembly Line Scheduling Problem. *IEEE Trans. Computers*, C-23(11):1169–1174, November 1974.

[139] Brent A. Kingsbury. The Network Queueing System, last visited April 1992. Available from http://power.curtin.edu.au/mirrors/nqs/Manuals/-Papers/MNQS/MNQS0001/MNQS0001.txt.

[140] D. Kirkham. Telstra's Experimental Broadband Network. *Telecommunications J. Australia*, 45(2), 1995.

[141] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, and M. E. Zosel. *The High Performance Fortran Handbook.* MIT Press, 1994. ISBN 0-262-11185-3.

[142] Hermann Kopetz. Scheduling. In Sape Mullender, editor, *Distributed Systems.* ACM Press, 1993. ISBN 0-201-62427-3.

[143] Boontee Kruatrachue and Ted Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, 5(1):23–32, January 1988.

[144] Cheolwhan Lee, Tao Yang, and Yuan-Fang Wang. Partitioning and Scheduling for Parallel Image Processing Operations. In *Proc. IEEE Symp. Parallel and Distributed Processing*, pages 86–90, October 1995.

[145] Michael Leventhal, David Lewis, and Matthew Fuchs. *Designing XML Internet Applications*. The Charles F. Goldfarb series on open information management. Prentice-Hall, Inc., 1998. ISBN 0-13-616822-1.

[146] A. Lewis, D. Abramson, R Sosic, and J. Giddy. Tool-based Parameterisation: An Application Perspective. In *Computational Techniques and Applications: CTAC95*. World Scientific, 1995.

[147] Ted G. Lewis and Hesham El-Rewini. *Introduction to Parallel Computing*, chapter Scheduling Parallel Programs. Prentice-Hall, Inc., Englewood Cliffs, New Yersey 07632, 1992. ISBN 0-13-498924-4.

[148] Sheng Liang. *The Java Native Interface Programmer's Guide and Specification*. The Java Series. Addison Wesley Longman, June 1999. ISBN 0-201-32577-2.

[149] David Lifka, Joseph Skovira, and Honbo Zhou. EASY-LL Administration Guide. Available from http://www.tc.cornell.edu/~lifka/easy-dist/Admin.html, Last visited July 1999.

[150] David A. Lifka. The ANL/IBM SP Scheduling System. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *LNCS*, pages 295–303. Springer, 1995.

[151] David A. Lifka, Mark W. Henderson, and Karen Rayl. Users Guide to the Argonne SP Scheduling System. Technical Report ANL/MCS-TM-201, Mathematics and Computer Science Division, Argonne National Laboratory, May 1995.

[152] B. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proc. SIGPLAN'88 Conf. Programming Language Design and Implementation*, pages 260–267, June 1988.

[153] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - a Hunter of Idle Workstations. In *Proc. IEEE 8th Int. Conf. Distributed Computing Systems*, pages 104–111, 1988.

[154] Virginia Mary Lo. Heuristic Algorithms for Task Assignment in Distributed Systems. *IEEE Trans. Computers*, 37(11):1384–1397, November 1988.

[155] LSC, Inc. SAM-FS. Available from http://www.lsci.com/lsci/products/-samfs.htm, Last visited May 1999.

[156] Brian A. Malloy, Errol L. Lloyd, and Mary Lou Soffa. Scheduling DAG's for Asynchronous Multiprocessor Execution. *IEEE Trans. Parallel and Distributed Systems*, 5(5):498–508, May 1994.

[157] J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and Development of Java Grande Benchmarks. In *Proc. of the ACM 1999 Java Grande Conference*, April 1999.

[158] Mauro Migliardi and Vaidy Sunderam. The Harness Metacomputing Framework. In *Proc. Ninth SIAM Conf. on Parallel Processing for Scientific Computing*, March 1999.

[159] Mauro Migliardi and Vaidy Sunderam. Heterogeneous Distributed Virtual Machines in the Harness Metacomputing Framework. In *Proc. Heterogeneous Computing Workshop of IPPS/SPDP 1999*, pages 60–73. IEEE Computer Society Press, April 1999.

[160] P. Mockapetris. Domain Names - Concepts and facilities. Request for Comments: 1034, available from ftp://ftp.is.co.za/rfc/rfc1034.txt, November 1987.

[161] P. Mockapetris. Domain Names - Implementation and specification. Request for Comments: 1035, available from ftp://ftp.is.co.za/rfc/rfc1035.txt, November 1987.

[162] R. Moore, D. Bender, C. Baru, J. Boisseau, W. Schroeder, J. Lopez, R. Marciano, T. Perrine, M. Gleicher, D. Nadeau, R. Frost, J. Litke, R. Klobuchar, D. Mosier, T. Lowman, D. Wade, A. Grimshaw, J. French, K. Marzullo, R. Wolski, J. Terstriep, R. Sharpe, D. Davis, J. Lindhei, G. Wheless, K. Lascara, and J. Kesselman. Research & Development Plan

and Schedule for the Distributed Object Computation Testbed (Intelligent Metacomputing Testbed). GA-C22507 sponsored by Advanced Research Projects Agency and U.S. Patent and Trademark Office, 1996.

[163] T. J. Mowbray and R. Zahavi. *The Essential CORBA: Systems Integration Using Distributed Objects.* Wiley, 1995. ISBN 0-471-10611-9.

[164] NASA Goddard Space Flight Center. Earth Observing System Data and Information System. Available from http://spsosun.gsfc.nasa.gov/-New_EOSDIS.html, last visited March 1999.

[165] National Center for Supercomputing Applications. Getting started with HDF - User Manual. University of Illinois at Urbana-Champaign, May 1993.

[166] National Key Centre for the Social Applications of Geographical Information Systems. Emergency Service System Response. Available from http://ch1.-gisca.adelaide.edu.au/kra/emsr.html, last visited March 1999.

[167] National Oceanic and Atmospheric Administration. NOAA Home Page. Available from http://www.noaa.gov, February 1999.

[168] Netscape Communications Corporation. CORBA/IIOP Developer Central. Available from http://developer.netscape.com/tech/corba/index.html, last visited March 1999.

[169] B. Clifford Neuman. Prospero: A Tool for Organizing Internet Resources. *Electronic Networking: Research, Applications and Policy*, 2(1):30–37, Spring 1992.

[170] B. Clifford Neuman, Steven Seger Augart, and Shantaprasad Upasani. Using Prospero to support integrated location independent computing. In *Proc. Symp. Mobile and Location Independent Computing*, August 1993.

[171] B. Clifford Neuman and Santosh Rao. Resource Management for Distributed Parallel Systems. In *Proc. 2nd Int. Symp. High Performance Distributed Computing*, pages 316–323, July 1993.

[172] B. Clifford Neuman and Santosh Rao. The Prospero Resource Manager: A Scalable Framework for Processor Allocation in Distributed Systems. *Concurrency: Practice and Experience*, 6(4):339–355, June 1994.

[173] B. Clifford Neuman and Theodore Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, 32(9):33–38, September 1994.

[174] Northeast Parallal Architectures Center at Syracuse University. WebSpace. Available from http://www.npac.syr.edu/projects/webspace/index.html, Last visited July 1999.

[175] Scott Oaks and Henry Wong. *Java Threads.* Nutshell Handbook. O'Reilly & Associates, Inc., United States of America, 1st edition, 1997. ISBN 1-56592-216-6.

[176] Object Management Group. CORBA/IIOP 2.2 Specification. Available from http://www.omg.org/corba/cichpter.html, July 1998.

[177] Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification (Revision 2.0). Framingham, MA, July 1995.

[178] On-Line Data Archives Program of the Advanced Computational Systems Cooperative Research Centre. Web page, last visited July 1999. Available from http://www.olda.dhpc.adelaide.edu.au.

[179] Open GIS Consortium. Web page. Available from http://www.opengis.org/, last visited March 1999.

[180] Open Software Foundation. *OSF DCE User's Guide and Reference.* Prentice-Hall, Inc., revision 1.0 edition, 1993. ISBN 0-13-643842-3.

[181] Open Software Foundation. *Introduction to OSF DCE.* Prentice-Hall, Inc., revision 1.1 edition, 1995. ISBN 0-13-185810-6.

[182] Oracle Corporation. Home Page. Available from http://www.oracle.com, last visited April 1999.

[183] Michael A. Palis, Jing-Chiou Liou, and David S. L. Wei. Task Clustering and Scheduling for Distributed Memory Parallel Architectures. *IEEE Trans. Parallel and Distributed Systems*, 7(1):46–55, January 1996.

[184] Craig J. Patten, K. A. Hawick, and F. A. Vaughan. A File System Layer for Flexible Bulk Data Transfer and Layout. Technical Report DHPC-039, Advanced Computational Systems CRC, Department of Computer Science, University of Adelaide, March 1998.

[185] Perihelion Software. *The Helios operating system.* Prentice Hall Int, 1989. ISBN 0-13-386004-3.

[186] Gregory F. Pfister. *In Search of Clusters.* Prentice-Hall, Inc., second edition, 1998. ISBN 0-13-899709-8.

[187] Platform Computing Corp. Load Sharing Facility. Available from http://www.-platform.com/, Last visited July 1999.

[188] Platform Computing Corporation. *LSF Administrator's Guide*, third edition, February 1996.

[189] Constantine D. Polychronopoulus and David J. Kuck. Guided Self-Scheduling: A Partical Scheduling Scheme for Parallel Supercomputers. *IEEE Trans. Computers*, C-36(12):1425–1439, December 1987.

[190] Spiridon Pulidas, Don Towsley, and John A. Stanovic. Imbedding Gradient Estimators in Load Balancing Algorithms. *IEEE 8th Int. Conf. Distributed Computing Systems*, pages 482–490, 1988.

[191] Ravi Ramamoorthi, Adam Rifkin, Boris Dimitrov, and K. Mani Chandy. A General Resource Reservation Framework for Scientific Computing. In *Proc. First Int. Scientific Computing in Object-Oriented Parallel Environments (ISCOPE) Conf*, December 1997.

[192] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed Resource Manafement for High Throughput Computing. In *Proc. Seventh IEEE Int. Symp. High Performance Distributed Computing*, July 1998.

[193] F. Ramme and K. Kremer. Scheduling a Metacomputer by an Implicit Voting Scheme. *Proc. 3rd IEEE Symp. High-Performance Distributed Computing*, pages 106–113, 1994.

[194] F. Ramme and T. Römke. The Computing Center Software A Step Towards Metacomputing. Technical Report Technical Report PC^2/TR-007-94, Paderborn Center for Parallel Computing, March 1994.

[195] F. Ramme, T. Römke, and K. Kremer. A Distributed Computing Center Software for the Efficient Use of Parallel Computer Systems. In *Proc. Int. Conf. HPCN Europe, vol II*, number 797 in LNCS, pages 129–136. Springer-Verlag, 1994.

[196] Friedhelm Ramme. Building a Virtual Machine Room – a Focal Point in Metacomputing. *Future Generation Computer Systems (FGCS)*, 11:477–489, 1995.

[197] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE.* O'Reilly & Associates, Inc., 1992.

[198] W. Rosenberry and J. Teague. *Distributing Applications Across DCE and Windows NT.* O'Reilly & Associates, Inc., 1993.

[199] András Salamon. DNS Resources Directory. Available from http://www.dns.-net/dnsrd/, Last visited July 1999.

[200] San Diego Supercomputer Center. Distributed Object Computation Testbed. Available from http://www.sdsc.edu/DOCT/, last visited July 1999.

[201] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* The MIT Press, Cambridge, MA, 1989.

[202] Vivek Sarkar and John Hennessy. Compile-time Partitioning and Scheduling of Parallel Programs. *Proc. SIGPLAN 86 Symp. Compiler Construction*, pages 17–26, 1986.

[203] Sanjeev K. Setia, Mark S. Squillante, and Satish K. Tripathi. Analysis of Processor Allocation in Multiprogrammed, Distributed-Memory Parallel Processing Systems. *IEEE Trans. Parallel and Distributed Systems*, 5(4):401–420, April 1994.

[204] Chien-Chung Shen and Wen-Hsiang Tsai. A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minmax Criterion. *IEEE Trans. Computers*, C-34(3):197–203, March 1985.

[205] Behrooz Shirazi, Hsing-Bung Chen, and Jeff Marquis. Comparative study of task duplication static scheduling versus clustering and non-clustering techniques. *Concurrency: Practice and Experience*, 7(5):371–389, August 1995.

[206] Behrooz Shirazi, Mingfang Wang, and Girish Pathak. Analysis and Evaluation of Heuristic Methods for Static Task Scheduling. *J. Parallel and Distributed Computing*, 10:222–232, 1990.

[207] Behrooz A. Shirazi, Ali R. Husson, and Krishna M. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems*, chapter Introduction to Scheduling and Load Balancing. IEEE Computer Society Press, Los Alamitos, CA, 1995. ISBN 0-8186-6587-4.

[208] Behrooz A. Shirazi, Ali R. Husson, and Krishna M. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems*, chapter Static Scheduling. IEEE Computer Society Press, Los Alamitos, CA, 1995. ISBN 0-8186-6587-4.

[209] Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load Distributing for Locally Distributed Systems. *Computer*, 25(12):33–44, December 1992.

[210] Joseph Skovira, Waiman Chan, and Honbo Zhou. The EASY - LoadLeveler API Project. In *Proc. IPPS 96*, 1996.

[211] Larry Smarr and Charles E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, June 1992.

[212] Warren Smith, Ian Foster, and Valerie Taylor. Predicting Application Run Times Using Historical Information. In *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[213] G. F. Stanlake. *Introductory Economics.* Longman Group Limited, London, 1967. SBN 582 350603.

[214] Ion Stoica, Hussein Abdel-Wahab, and Alex Pothen. A Microeconomic Scheduler for Parallel Computers. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *LNCS*, pages 200–218. Springer, 1995.

[215] Harold S. Stone. Multiprocessor Scheduling with the Aid of Network Flow Algorithms. *IEEE Trans. Software Engineering*, SE-3:85–93, January 1977.

[216] Sun Microsystems. Network File System Version 3 (NFSv3) Specification. RFC 1813, June 1995.

[217] Sun Microsystems. Java Advanced Imaging API White Paper (EA2). Available from http://java.sun.com/products/java-media/jai/, November 1998.

[218] Sun Microsystems. Java Products Homepage. Available from http://www.-javasoft.com/products/, last visited July 1999.

[219] Sun Microsystems. Java Web Server. Available from http://www.sun.com/-980310/javawebserver/, Last visited July 1999.

[220] Supercomputer Computations Research Institute. Distributed Queueing System 3.1.3 Reference Manual. Florida State University, Tallahassee, Florida, March 1996.

[221] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, 3rd edition, 1996. ISBN 0-133-9428-1.

[222] The Australia New Zealand Land Information Council (ANZLIC). Core Metadata Elements for Land and Geographic Directories in Australia and New Zealand. Available from http://www.anzlic.org.au/metaelem.htm, last visited March 1999.

[223] The Federal Geographic Data Committee. The National Spatial Data Infrastructure. Available from http://www.fgdc.gov/NSDI/Nsdi.html, last visited March 1999.

[224] Don Towsley. Allocating Programs Containing Branches and Loops Within a Multiple Processor System. *IEEE Trans. Software Engineering,* SE-12(10):1018–1024, October 1986.

[225] U.S. Geological Survey's (USGS) EROS Data Cente. Multispectral Scanner Landsat Data. Available from http://edcwww.cr.usgs.gov/glis/hyper/guide/-landsat, last visited March 1999.

[226] U.S. National Imagery and Mapping Association. U.S. Imagery and Geospatial Information System (USIGS) Architecture Framework. Available from http://-www.nima.mil/aig/products/uaf/, November 1997.

[227] U.S. National Imagery and Mapping Association. U.S. Imagery and Geospatial Information System (USIGS) Technical Architecture. Available from http://-www.nima.mil/aig/products/uta/, November 1997.

[228] U.S. National Imagery and Mapping Association. USIGS Geospatial and Imagery Access Services (GIAS) Specification, version 3.1, N0101-B. Available from http://www.nima.mil/aig/products/uip/gias/, February 1998.

[229] U.S. National Imagery and Mapping Association. Web page. Available from http://www.nima.mil/, last visited March 1999.

[230] Gregor von Laszewski and Ian Foster. Usage of LDAP in Globus. Available from http://www.globus.org, 1998.

[231] Edward F. Walker, Richard Floyd, and Paul Neves. Asynchronous Remote Operation Execution in Distributed Systems. In *Proc. Tenth Int. Conf. Distributed Computing Systems*, pages 253–259, May 1990.

[232] Larry Wall and Randal L. Schwartz. *Programming Perl.* O'Reilly & Associates, January 1991. ISBN 0-937175-64-1.

[233] Jon B. Weissman and Andrew S. Grimshaw. A framework for partitioning parallel computations in heterogeneous environments. *Concurrency: Practice and Experience*, 7(5):455–478, August 1995.

[234] Roy Williams. SARA: The Synthetic Aperture Radar Atlas. Available from http://www.cacr.caltech.edu/~roy/sara/, last visited February 1999.

[235] Roy Williams and Bruce Sears. A High-Performance Active Digital Library. In L. O. Herzberger and P. M. A. Sloot, editors, *Proc. HPCN98*, Lecture Notes on Computer Science. Springer, April 1998.

[236] Patrick Henry Winston. *Artificial Intelligence.* Addison-Wesley, third edition, 1992. ISBN 0-201-53377-4.

[237] R. Wolski. Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service. In *Proc. 6th IEEE Symp. High Performance Distributed Computing*, January 1997.

[238] World Wide Web Consortium. Extensible Markup Language (XML). Available from http://www.w3c.org/XML.

[239] Tao Yang, Cong Fu, Apostolos Gerasoulis, and Vivek Sakar. Mapping Iterative Task Graphs on Distributed Memory Machines. In *Proc. 24th Int. Conf. Parallel Processing*, volume II, pages 151–158, 1995.

[240] Tao Yang and Apostolos Gerasoulis. DSC: Scheduling Parallel Tasks in an Unbounded Number of Processors. *IEEE Trans. Parallel and Distributed Systems*, 5(9):951–967, September 1994.

[241] Tao Yang and Oscar H. Ibarra. On Symbolic Scheduling and Parallel Complexity of Loops. In *Proc. IEEE Symp. Parallel and Distributed Processing*, pages 360–367, October 1995.

[242] Honbo Zhou. Scheduling DAGs on a bounded number of processors - A new approach. In *Proc. Int. Conf. Parallel and Distributed Processing, Techniques and Applications*, August 1996.