Resource Provision in Object Oriented Distributed Systems

Stephen W. Proctor

Ph.D.

University of Edinburgh

1989

Abstract

Using objects to structure distributed systems is becoming an increasingly popular paradigm. This thesis examines some of the fundamental problems associated with resource provision in such systems. A conceptual framework for the work is created by the development of a reference model for object oriented distributed systems. Within this framework, several aspects of resource provision are examined in detail. In each case, an object oriented solution is sought rather than applying existing, process based solutions.

The problem of object construction in a heterogeneous environment is addressed, leading to the development of a distributed transformation algorithm for the automatic construction of object representations. A novel scheduling mechanism is developed based upon statistical hypothesis testing. Two applications of this mechanism are simulated in detail : the assignment of invocation messages to object instances, and the suppression of redundant status update messages. The concept of 'virtual properties' is introduced, leading to the development of virtual templates as a re-usable mechanism for endowing objects with properties such as resilience and persistence. The separate resource provision issues addressed are then drawn together to demonstrate how the techniques developed can be used to satisfy users' resource requirements.

Acknowledgements[1]

Although only one name appears on the front page, completing a PhD is a task that requires support from many people. First and foremost I would like to thank Gordon Brebner for his excellent supervision. Gordon's thought provoking conversation and his (frustrating) ability to always provide a counter-example, have been invaluable in developing the work presented here. Thanks also to Rob Pooley for patiently answering my many questions on programming in Simula.

Many thanks to Alistair, Tom and Tim who, over the years, have shared an office with me, helping to keep me sane. Thanks also to everyone at Crew House for making my stay in Edinburgh such a pleasant one, especially to Roger for keeping me fit; to David for the games of squash; to Michelle for the cups of tea; and to Dave just for being Dave.

Finally, I would like to thank Russell for inspiring me to start this course; and most of all, thankyou to my fiancé, Nicola, for inspiring me to finish.

---

[1]This work was funded by the Science and Engineering Research Council. Thanks are also due to my employer, British Telecom, for allowing me the time to perform these studies.

Declaration

I declare that this thesis was composed by myself and that the work it describes is my own, except where stated in the text.

To my first teachers, Sylvia and Arthur Proctor.

Table of Contents

List of Figures

List of Tables

Chapter 1

Object Oriented Distributed Systems

The phrase 'object oriented distributed system' is ambiguous as both 'object orientation' and 'distribution' have many interpretations. This chapter provides an introduction to both areas, establishing the interpretations assumed throughout this thesis. The range of distribution for the three fundamental components of hardware, data and control are examined. Several typical configurations are identified, ranging from closely coupled parallel architectures to wide area networks. The flavour of distribution to be addressed is characterized by personal workstations connected via a local area network. The key features of object orientation : invocation, encapsulation, and inheritance are defined. An overview is then presented of six example object oriented distributed systems. The chapter is completed with a description of the research aims and an overview of the remaining chapters.

1.1 Distributed Systems

In computing literature the term **Distributed** is applied to a wide range of multi-computer and multiprocessor systems. The lack of standard vocabulary to describe the many flavours of distribution invariably results in ambiguity. Every author, including this one, must re-define 'distributed' to suit the particular system under discussion. This problem has long been recognised [Enslo78], [LeLan81], but all attempts to define a standard terminology have, inevitably, failed. In this opening section various aspects of distribution are examined. From this discussion will emerge the flavour of distributed system addressed throughout this thesis.

1.1.1 What is Distributed?

There are many aspects of computer systems that can be 'distributed'. The three fundamental components are :

- Hardware
- Data
- Control

The following discussion examines the range of distribution possible for each of these components.

Distributed Hardware

The simplest example of hardware distribution is a **Single Processor**. The term *processor* is used here to describe a self-contained, independent computer; for

example a processing element (CPU) capable of executing instructions, memory for storing data, and peripherals such as a keyboard and screen. The level of distribution in a processor spans no further than a single circuit board or perhaps a collection of circuit boards within a single cabinet. Within the context of this thesis single processors are therefore considered non-distributed. Examples include desktop personal computers through to large mainframes. Moving slightly into the realms of distribution are **Shared Memory Multiprocessors**. In these systems, the processing hardware is replicated to allow parallel processing, but the memory is shared. Interprocessor communication is relatively easy in such systems, utilising shared access to data held in common memory. This configuration is typical of many modern parallel processors.

Multiple Processor systems are more distributed still. In these systems single processors are interconnected by some form of communication channels. There is no shared memory, interprocessor communication being performed by passing messages along the communication channels. The processors may be distributed to varying degrees, loosely measured by the channel 'length'. For example, messages may pass between processors on the same circuit board, alternatively they may travel up to a few thousand metres via a **Local Area Network** (LAN) or, ultimately, they may travel thousands of miles through a **Wide Area Network** (WAN).

Distributed Data

Non-distributed data occurs when only a **Single Copy** of the data exists throughout the whole system. **Full Replication** of data is the simplest level of distribution in which multiple, complete copies of the data are made. With full replication, modifications to any one copy of the data must also be applied to the others if data consistency problems are to be avoided. A greater level of distribution is provided by **Simple Partitioning**, where a single copy of the data is partitioned, each partition being located with a different processor. Consistency problems are

thus avoided since there is only one copy of each partition. Simple Partitioning can be extended to **Redundant Partitioning** by distributing multiple copies of each partition. Here again the multiple copies of each partition must be coordinated with regard to modifications, in order to maintain consistency.

Distributed Control

In the case of a single processing element there can only be one point of control and therefore the question of control distribution does not arise. Assuming more than one processing element, there are several possible control configurations. The simplest, i.e., non-distributed case is a strict master/slave relationship, where one processor dictates the actions of all others on a step by step basis. Control is therefore **Centralised** at the master processor.

Control becomes more distributed when each processor is allowed **Complete Autonomy** to work on independent tasks. This allows parallel processing of independent tasks, but there is no coordination between processors that would allow cooperation on common tasks. Adding cooperative behaviour yields **Multiple Co-operating Control Points**. Under this configuration processors work together, each performing a sub-task that forms part of a larger, common task. Although control over each sub-task is centralised at the processor performing the sub-task, the processors amalgamate their individual contributions in order to solve the common task. Hence, control of the common task is fully distributed.

Enslow proposed that the three components of hardware, data and control be represented by three orthogonal axes [Enslo78]. Each axis is labelled with discrete categories, moving from completely centralised at the origin through several stages to completely distributed. This gives a 'classification cube', an example of which is shown in Figure 1–1. The cube's axes have been labelled to make them consistent with the terminology introduced earlier. Distributed systems can be mapped into

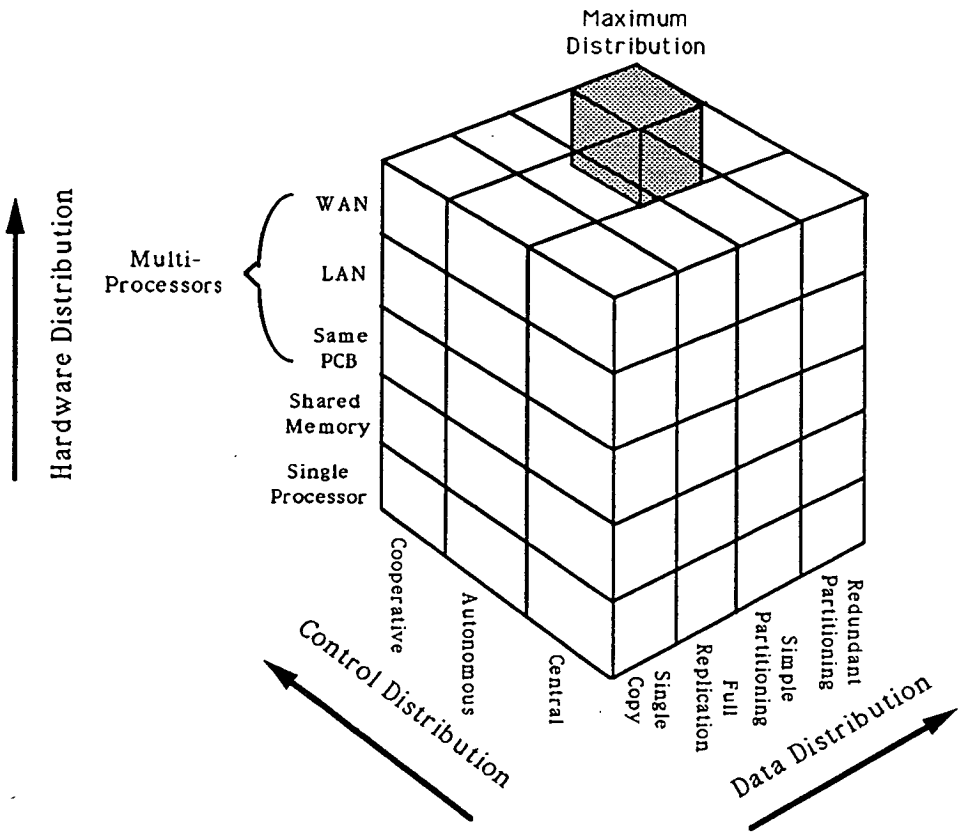

Figure 1–1: Enslow's Distribution Classification Cube (modified)

a segment of the cube according to the extent of their hardware, data and control distribution.

1.1.2 Examples

The classification cube provides a simple measure of the level of distribution within a system. Although in principle all combinations of hardware, data and control distribution are possible, in practice several particular combinations predominate.

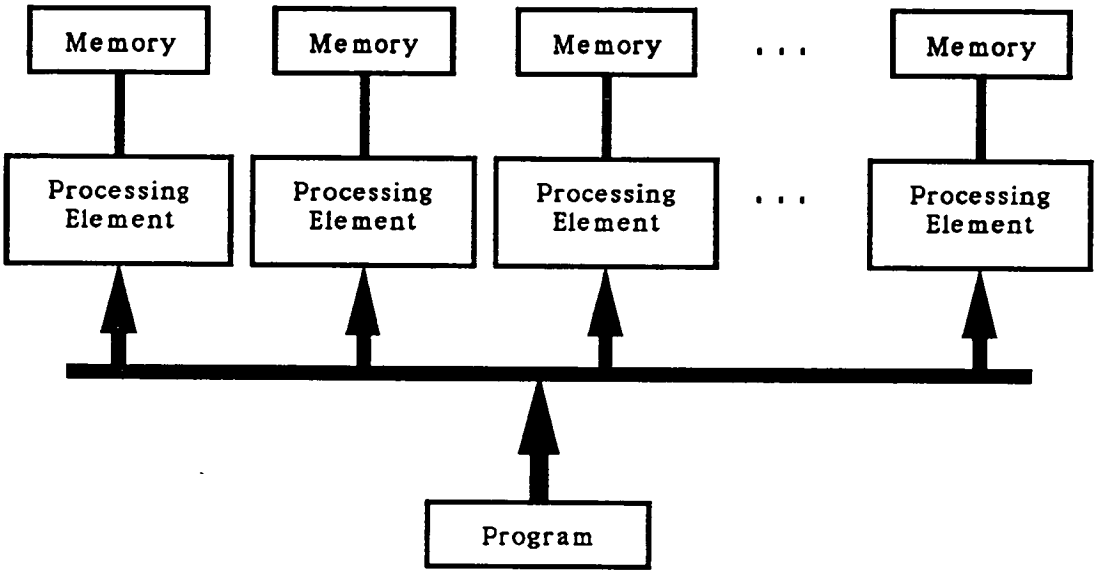

Figure 1–2: A typical Array Processor

Array Processors

An array processor consists of many (possibly hundreds) of processing elements, each with their own dedicated memory (Figure 1–2). Data is partially replicated, with each memory unit typically containing an element from a large array. Every processing element executes the same instruction at the same time, controlled by a central program, i.e., there is no control distribution. Hence, array processors perform operations on an entire array in a single step. A more detailed analysis of array processing can be found in [Hwang85].

Closely-Coupled Systems

Closely-coupled, or tightly-coupled systems consist of multiple processing elements accessing a common memory (Figure 1–3). This configuration supports single copies of data items, i.e., no data distribution. Control distribution could be completely autonomous, however, the shared memory facility provides a relatively

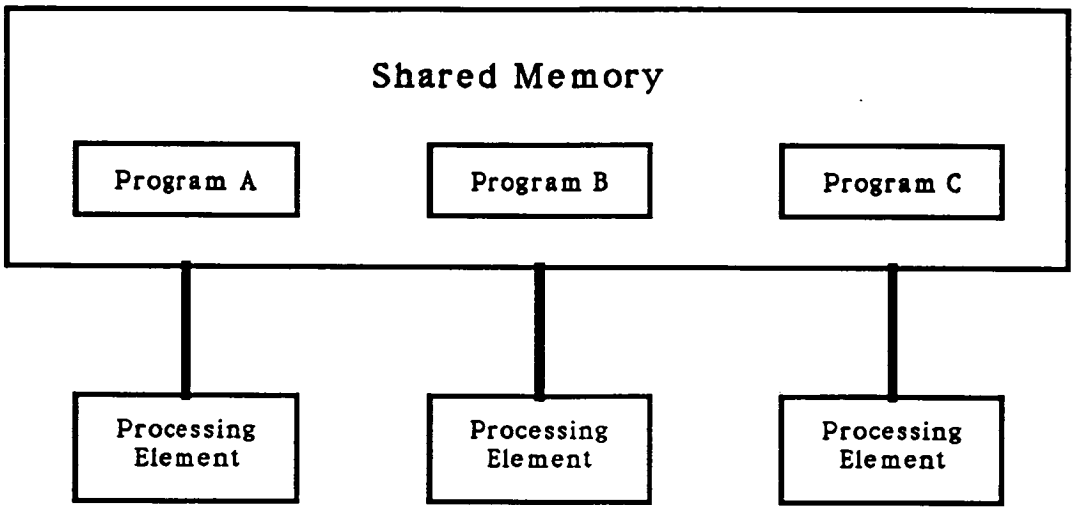

Figure 1–3: A Closely-Coupled Multiprocessor System

inexpensive program interaction mechanism, encouraging fully cooperative control. [Hwang85] provides further details on closely-coupled systems.

Loosely-Coupled Systems

Loosely-coupled systems are characterized by self contained processors interconnected by a local area network (Figure 1–4). Personal workstations connected by Ethernet provide a typical example. The degree to which data and control are distributed depends upon the nature of the operating system built upon this hardware base. **Network Operating Systems** typically encourage fully replicated data and complete autonomy. **Distributed Operating Systems** typically provide partially replicated data and cooperative control. A more detailed examination of these operating system types is presented in section 1.5

Wide-Area Systems

Wide area systems consist of geographically dispersed processors connected by wide area networks (Figure 1–5). The distance between adjacent processors may

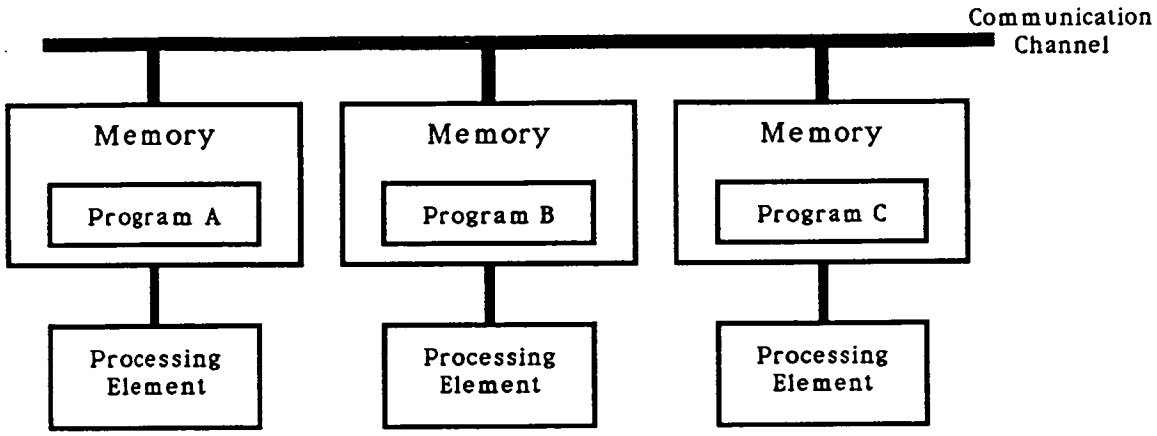

Figure 1–4: A Loosely-Coupled Multiprocessor Distributed System

span national and even international borders. Such systems exhibit autonomous control and fully replicated data. Many examples are identified in [Quart86], including ARPANET [Tane81a] and JANET [Patel88].

1.2 System Definition

This thesis concentrates on loosely-coupled distributed systems, as depicted in Figure 1–4. A detailed definition of the environment addressed is presented in chapter 3. For the moment it will be defined simply as consisting of processors connected by a local area network. Henceforth the prefix 'loosely-coupled' will be dropped, and the phrase 'distributed system' will be understood to imply 'loosely-coupled distributed system'.

1.3 Benefits of Distribution

Distributed systems offer potential for many improvements upon non-distributed systems. Some of these benefits are examined below.

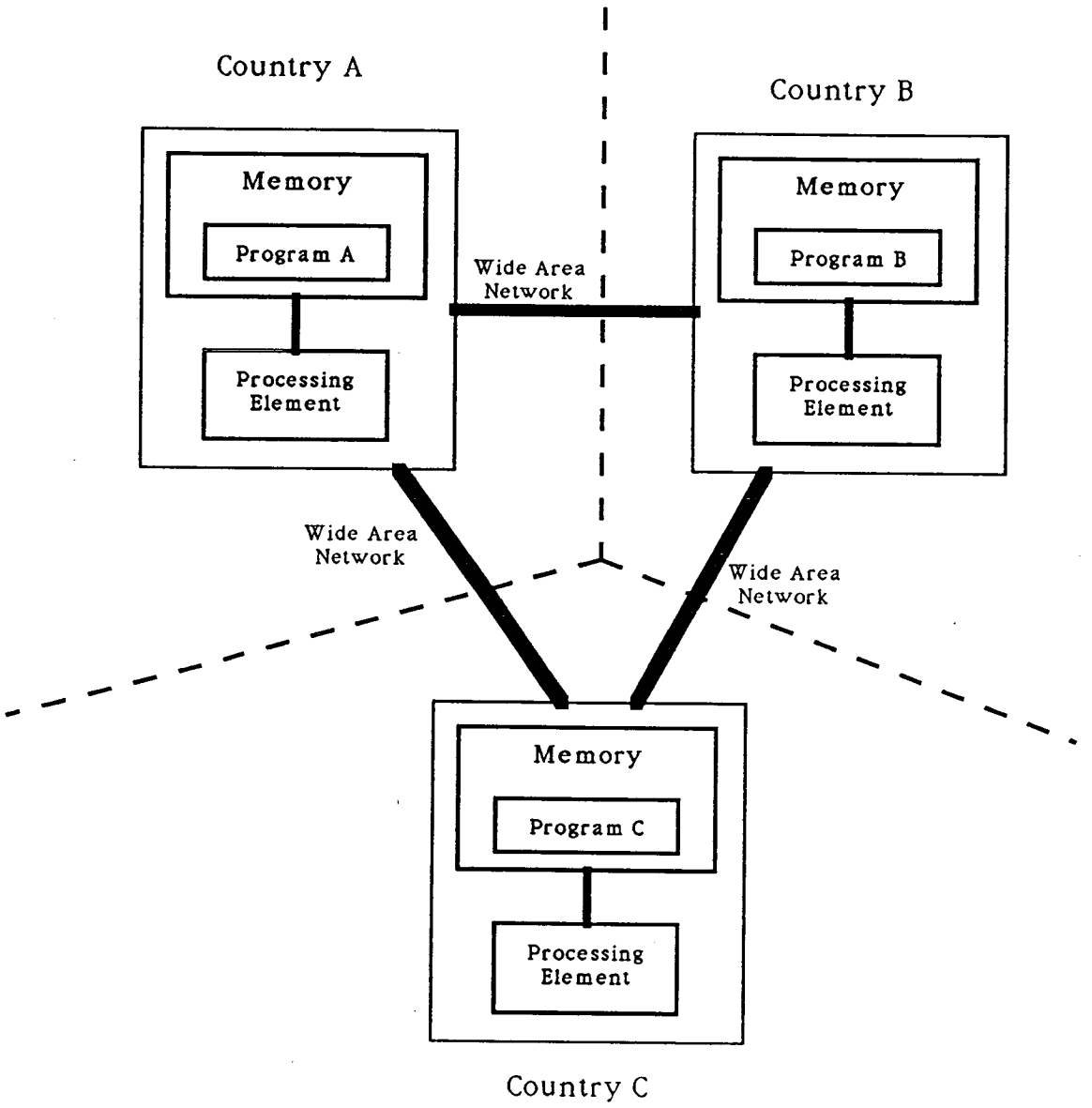

Figure 1–5: A Wide-Area System

1.3.1 Improved Performance

The multiple processors of a distributed system allow multiple concurrent threads of execution to exist, i.e. **Parallel Processing**. Autonomous control can be used to perform independent tasks at different processors. This provides parallelism at the system level, but not at the user level. In order to provide parallel processing of individual users' work, multiple cooperative control is required, with each control point operating on partially replicated data towards a coordinated common goal.

There is further scope for performance improvement by assigning tasks to 'suitable' processors. The principle of selectively matching tasks and processors lies at the heart of **Distributed Scheduling**, a subject dealt with extensively later in the thesis. At present it suffices to state the basic motivation behind distributed scheduling, which is to make the most efficient use of available resources.

1.3.2 Configuration Flexibility

In principle, the resources provided by a distributed system can be increased simply by adding further processors to the network, causing little or no disruption to the existing system's operation. The ease with which a distributed system can be extended (or reduced) is a direct result of its distributed nature. By contrast, expanding the capabilities of a centralised processor implies replacing it with something else.

The flexibility afforded by a distributed environment encourages the use of specialised processors for specialised tasks. Any task capable of utilising a specialised processor should automatically be directed there by the system. This is just one example of distributed scheduling as defined earlier. Specialised processors introduce heterogeneity into the distributed environment. Systems in which the processors are not all identical are generally termed **Heterogeneous**. **Homogeneous** distributed systems, i.e., where all processors *are* identical, retain the physical ex-

tensibility mentioned above, but can not exploit the functional flexibility created by specialist hardware.

1.3.3 Reliability

By providing redundant hardware and data, distributed systems can mask occasional processor failures. When a failure occurs, the failed processor's tasks can be restarted on other, 'spare' processors. The system's users remain unaware of the failure perceiving only, perhaps, a slightly longer execution time due to the re-start.

Some systems provide a **Checkpoint** mechanism whereby a processor can periodically save the current state of its executing tasks. Under these circumstances a failure does not require a complete re-start. Each task can be re-started from its latest checkpoint information, thereby avoiding the repetition of earlier processing. This technique is known as **Rollback Recovery.**

1.3.4 Resource Sharing

Attaching all processors to a common network provides shared access to the system's resources. For example, expensive specialised processors such as printers and disc drives can be shared equally between many users.

On a more general level, users of a distributed system have access to the entire collection of resources the system provides. Users can therefore be allocated resources according to their needs. For example, given ten users and ten processors the optimal allocation may not be one machine per user. If one user generates as many tasks as the other nine combined, then the optimal configuration may be five processors for the large user and five processors shared between the remaining nine users. This form of resource sharing, generally termed **Load Balancing**, is another example of distributed scheduling.

1.3.5 Reduced Cost

Historically, the price-performance ratio for computer hardware has favoured large, stand-alone machines over multiple, smaller machines. This observation was enshrined in *Grosch's Law* which stated that

> "The processing capacity of a computer system is roughly proportional to the square of its cost."

However, technological advances in computer hardware are continually improving processor performance, while also reducing costs. This has invalidated Grosch's law, changing the price-performance ratio to favour multiple, low-performance processors. As a result, distributed systems have become economically viable when compared to large mainframes.

1.4 Disadvantages of Distribution

The benefits noted above are not without cost. Distributed hardware, data and control introduce significant problems for the system designer. In general, the cost is one of greater complexity.

Improved performance, created by utilising multiple cooperative control points to perform parallel processing, is gained at the expense of greater overheads in coordinating the parallel activities. Configuration flexibility, which encourages heterogeneity, requires multiple program representations, one for each heterogeneous processor type. Reliability through replication consumes resources since the redundancy created to increase reliability, ties up resources that could have been used for other services. Finally, sharing of resources requires careful access management. Resource usage must be monitored, with scheduling employed to avoid resource conflicts and ensure 'fairness'.

1.5 Distribution and Operating Systems

The operating system's role is the same for both distributed and non-distributed environments; namely, to manage the system's resources. There are two basic approaches to operating system design in a distributed environment. Systems in which the network is visible to users are termed **Network Operating Systems.** Systems that hide the network, attempting to present distributed resources as one large, uniform processor, are termed **Distributed Operating Systems.** These two philosophies are examined below.

1.5.1 Network Operating Systems

Network operating systems are generally implemented as a collection of programs that run on a processor's resident, non-distributed operating system. These programs allow users, for example, to log in to remote processors, run programs on remote processors and copy data between processors. Utilising existing, non-distributed operating systems in this way makes network operating systems relatively easy to implement, but has been likened by Tanenbaum to

> "... tying together a collection of incompatible [processors] with bailing wire and bubble gum." [Tane81a]

The major drawback to network operating systems is the need for users to be aware of, and understand, the system's configuration. For example, to issue a command such as

RUN COMMAND Y AT PROCESSOR X

the user must be aware both that processor X exists and that it is capable of running command Y. The fact that these assignments are explicitly under user

control denies the system the chance to optimise resource usage. For example, processor X may be heavily overloaded whilst, unknown to the user, processor W is sitting idle.

1.5.2 Distributed Operating Systems

Distributed operating systems attempt to present distributed resources as part of a single, uniform processor. Items such as programs and data have logical names that are location independent. The operating system automatically translates between the logical name and the actual, location dependent name. This decoupling between user name space and system name space gives the distributed operating system complete control over the placement of tasks and data. For example, users can now issue commands such as

RUN COMMAND Y

It becomes the operating system's responsibility to find a suitable processor on which to run the command. Users perceive all commands as being run locally and remain unaware of processor boundaries. Tanenbaum summarised this concept with the rule of thumb that

> "If you can tell which [processor] you are using, you are not using a distributed system." [Tanen85]

The expense associated with this improved environment is greater implementation complexity. The programming and design effort involved in creating a distributed operating system is considerably greater than that for a network operating system. This is particularly true if the distributed operating system is developed from scratch rather than building upon an existing system. For this reason, most extant distributed operating systems use an existing, non-distributed operating system as a foundation. UNIX is typically used to fulfil this role.

1.6 Object Orientation

In common with the phrase 'distributed system', object orientation has many interpretations [Strou88]. Rentsch predicted with reasonable accuracy that

> "Object oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favour of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no-one will know just what it is." [Rents82]

This section attempts to convey the general philosophy behind object orientation. Details specific to the environment addressed by this thesis are given in chapter 3.

1.6.1 Objects

An object is a repository for **Data** supported by a collection of **Procedures** to manipulate this data. The data is private to the object and can only be accessed indirectly by requesting the object to invoke one of its procedures. These procedures, known variously as **Invocation Routines**, **Methods**, **Actors**, **Behaviours** or **Operations**, are the only externally visible attributes an object possesses (Figure 1–6).

Every object is defined by its **Class**, alternatively referred to as its **Type**. A **Class Definition** (implementation) is a functional description of the object's procedures and the data they operate on. It represents the blueprint from which **Instances** of the class are created. Programmers using object oriented languages create classes, not objects. Once a class is defined it can be used to create an arbitrary number of instances. Every instance of a class has identical structure,

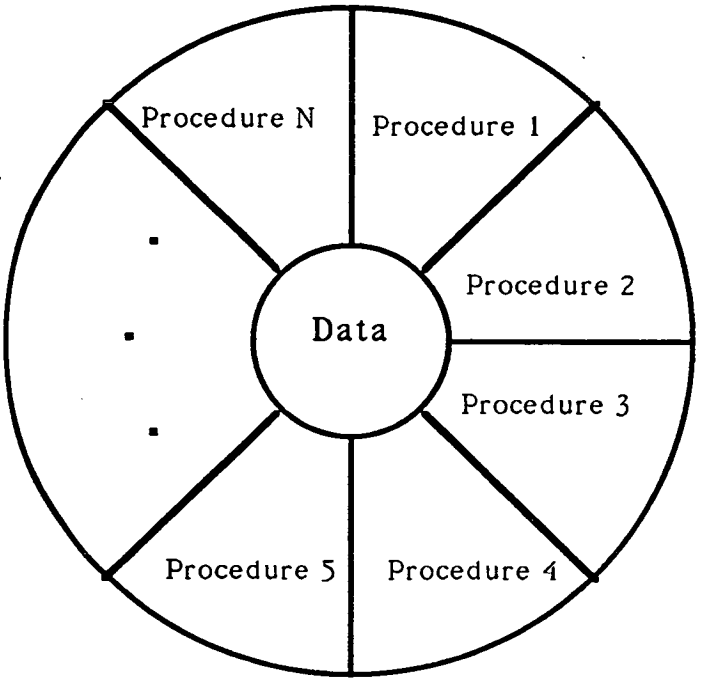

Figure 1–6: The Logical Structure of an Object

but the values assigned to the data they contain may be different. For example, every instance of class `Thesis` has a title, an abstract, and pages of contents (Figure 1–7). However, once these data items have been suitably initialised (using the appropriate procedures from the `Thesis` class definition) each instance of `Thesis` will be unique. Subsequent requests to `showAbstract` or `showPage` (also in the `Thesis` class definition) will yield different results depending upon which `Thesis` instance the request is directed to.

1.6.2 Invocation

In traditional procedural programming languages, a call to a sub-routine or a library function explicitly references the code to be executed. The calling program therefore decides not only which service to call, but also nominates the code to perform the service.

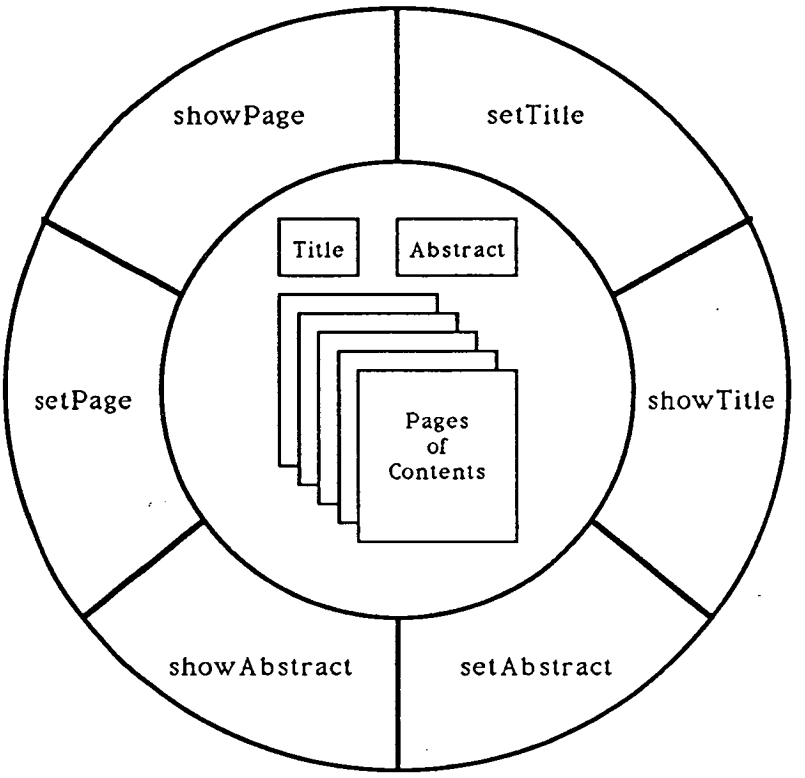

Figure 1–7: A Thesis Object Instance

sendTo:	service:	parameters:	replyTo:	systemSpecific:
myThesis	setPage	pageNo Page of Text	userSWP	Access Rights

Figure 1–8: An Example Invocation Message

In object oriented systems, requests to perform a service are passed to object instances via **Invocation Messages.** An invocation message, generated by the object requesting the service, contains the name of the object to be invoked and the name of the service required. It also contains the parameters the service requires, the name of the requesting object and, optionally, may include system specific information such as the requester's access privileges. An example invocation message is shown in Figure 1–8. The requesting object's name is used as the reply address should the invocation generate a reply. Invocation and reply messages are the only legal form of interaction in object oriented systems.

The service name specified in an invocation message is known as the **Message Selector.** Each object contains a **Message Handler** that examines incoming messages, assigning requests to the appropriate procedures (Figure 1–9). This mechanism de-couples the *what* from the *how.* The invocation message specifies *what* is required, but the invoked object decides *how* it should be done.

1.6.3 Specification and Implementation

An object was characterized in section 1.6.1 as consisting of data and procedures. We can now improve this characterization by stating that an object's specification is defined exclusively by the invocation messages it understands. An implementa-

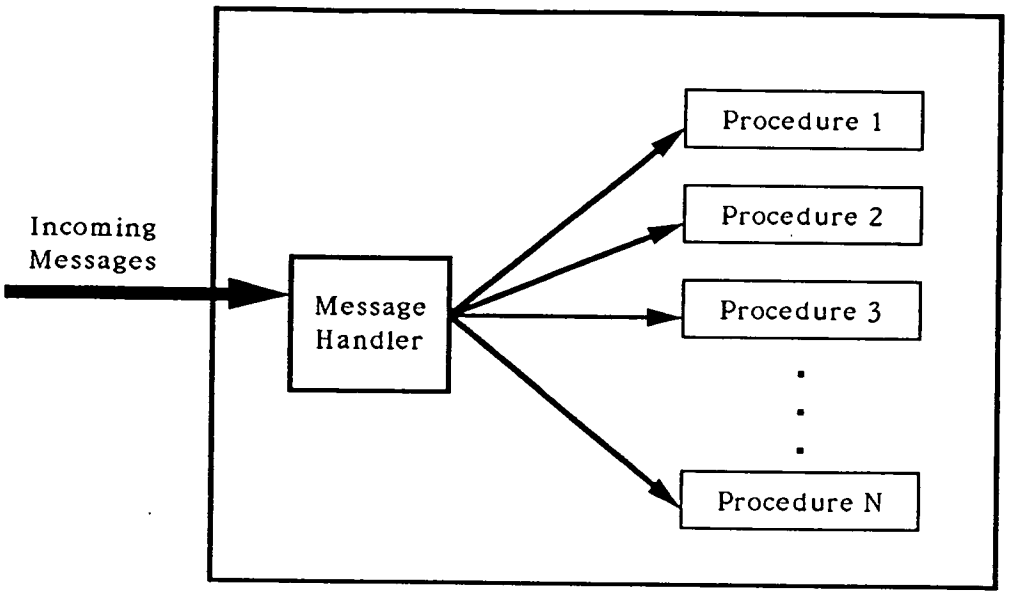

Figure 1–9: Invocation Message Handling

tion is said to satisfy a specification if it is capable of responding to all invocation messages the specification defines. An object's specification and implementation are known as its **Abstract Type** and **Concrete Type** respectively.

Note that a specification can be satisfied by many different implementations. Multiple implementations of the same specification are said to be **Functionally Compatible** or **Type Compatible**, because they share the same abstract type. Note also that, by this definition, an implementation supporting a super-set of the messages defined by an abstract type is also type compatible.

1.6.4 Encapsulation

Using message selectors, a requesting object specifies *what* service is required. The invoked object, using its message handler, has full responsibility for deciding *how* that service should be provided. This apparently trivial feature, i.e., adding a level of indirection to the invocation mechanism, is the key to the power behind object orientation because it encourages encapsulation.

Encapsulation minimises the interdependencies among separately implemented objects by defining strict external interfaces [Snyde86]. The initiator of an invocation message requires no knowledge of how the object performs the service requested. Indeed, if encapsulation is enforced rigorously then invokers are actively forbidden such knowledge. Hence, an object's implementation and the arrangement of its internal data are encapsulated in a procedural shell that mediates all access to the object. Encapsulation therefore enforces well established software engineering techniques such as problem partitioning, information hiding and abstract data typing.

1.6.5 Inheritance

Inheritance has been claimed as the distinguishing feature between languages supporting data abstraction, and languages supporting object orientation [Strou88]. It provides a mechanism for organising, building and using reusable classes [Halbe87]. With inheritance, new classes are defined in terms of existing classes by stating how the new class differs from the old. Inheritance therefore promotes development by refinement.

If a class c directly inherits from a class p then p is a **Parent** of c and c is a **Child** of p. The parent and child classes are sometimes referred to as the **Super-Class** and **Sub-Class** respectively. Inheritance creates a **Class Hierarchy** or **Type Hierarchy** in which the terms **Ancestor** and **Descendant** are applied in the usual sense.

A sub-class inherits the external interface and concrete type of its parent-class, which in turn inherited from *its* parent-class, and so on. Hence, a sub-class automatically satisfies the cumulative specification of all its ancestors. The new class is distinguished by the addition of further procedures specific to itself. These new procedures are automatically available to any future descendants of the new class. By encouraging the reuse of existing code, inheritance considerably reduces the

development effort required to create new classes. In a mature object oriented environment there will be an existing class possessing many of the attributes required by the new class. The new class can simply be declared as a child of this similar class, inheriting its implementation. The differences between the new class and its parent are then defined. Some languages allow **Multiple Inheritance** so that an object inherits the implementation of two or more parent classes. These are merged together, with suitable additions, to define the new class.

1.7 Objects and Distributed Systems

Object instances are self contained units that can, in theory, be arbitrarily located within a distributed environment. The hidden implementation of objects can be used to mask heterogeneity; each different processor having its own concrete type satisfying the object's abstract type. Invocation messages are self contained requests for service that can be passed across a network just as easily as they can be passed between objects on the same processor. Hence, encapsulation, coupled with invocation messages, makes object orientation an excellent paradigm for structuring distributed software. In particular, object orientation is an excellent paradigm for constructing distributed operating systems [Jones79].

An **Object Oriented Distributed System** is a loosely coupled distributed environment, supported by a distributed operating system whose design is based upon the object paradigm.

1.8 Example Object Oriented Distributed Systems

The following examples have been selected to collectively illustrate the major research issues tackled by current designs. The intention is to introduce the design problems faced when building such systems. Each example has its own specialisation that will be used throughout the thesis to illustrate certain points. In particular, chapter 2, which describes the Object Reference Model, will make comparisons between these examples.

1.8.1 Amoeba

Amoeba is a research project on distributed operating systems being carried out at the Vrije Universiteit in Amsterdam. Its goal is to investigate capability-based, object-oriented systems, and to build a working prototype system to use and evaluate [Tane81b], [Tanen86]. Amoeba's designers claim it to be the "World's Fastest Distributed Operating System" [Renes88].

The Amoeba architecture consists of four principal components.

- Workstations, one per user. At present, SUN-3's are used.
- Pool processors, where most of the computing occurs. These are not assigned permanently to any individual user, but are allocated and deallocated dynamically as required.
- Specialised servers such as file servers, directory servers and data base servers.
- Gateways used to link Amoeba systems at different sites.

All Amoeba machines run the same kernel, which primarily provides message-passing services. The gateways connect Amoeba systems at different sites, potentially in different countries, into a single, uniform system.

Objects in Amoeba are named using *capabilities.* A capability is an 'authorisation token' giving its holder authority to invoke a particular service. An appropriate capability must be included as part of every invocation message. Access security is further supported by capability encryption and the use of sparse address ports. The address port used to communicate invocations to an object is chosen randomly from a very large address space. The design is such that the probability of correctly guessing a valid port address is very small. Port numbers do not imply a specific location. If an object migrates to another location it carries its port with it. The invoker of an object must therefore possess the correct port number and a valid capability in order to be successful.

As with most current systems there is a strong tie-in with UNIX. A UNIX emulation environment has been created on top of Amoeba to allow the porting of UNIX software such as shells, editors, compilers and standard utilities.

1.8.2 Eden

The Eden project [Almes85], [Lazow81], [Almes87], is an experiment by the Department of Computer Science at the University of Washington, Seattle in designing, building and using an "integrated distributed" computing system. It :

> "Attempts to combine the benefits of integration and distribution by supporting an object based style of programming on a collection of node machines connected by a local network." [Almes85]

Eden objects, called *Ejects*, are programmed using the Eden programming language (EPL), [Black85]. EPL supports concurrency within Ejects, presenting

Eject programmers with the illusion of multiple threads of control. Eden supports a checkpoint operation that when invoked by an Eject creates a passive representation of the Eject — a data structure designed to endure system crashes. The data in a passive representation is sufficient to enable the Eject to reconstruct its long term state in the event of a processor failure.

Eject naming and access control are performed using capabilities. Eject capabilities contain a unique Eject identifier and a set of access rights granted to the capability's holder. Invocations are made by presenting the system with an Eject capability, the symbolic name of the operation to be performed and any parameters required. Given an Eject capability, the Eden kernel is responsible for locating the invoked object.

The Eden kernel operates on a collection of VAX/UNIX systems interconnected by Ethernet [Almes85], and on a network of Ethernet connected Sun workstations [Almes87]. In both cases, Eden is built upon and co-exists with the host's UNIX operating system.

1.8.3 Clouds

The Clouds project began in 1981 at Georgia Tech, Atlanta, to develop a fault tolerant distributed operating system [Dasgu86]. Fault tolerance, also known as resilience, is the ability to provide a reliable service based upon unreliable hardware.

Objects in the Clouds system are persistent. A persistent object is one whose internal state is non-volatile and once created remains in the system until explicitly deleted. This gives object state information the same status as other, long-lived data that is usually stored explicitly in files.

Fault tolerance is provided using a primary-backup technique. Each fault-tolerant object has associated with it a backup object located on a different pro-

cessor. The backup has the same facilities as the primary, but remains dormant most of the time. Periodically the backup probes the primary to make sure it is still active. If the backup detects that the primary has failed then it takes over as the primary object.

Atomic Transactions are used to ensure that failures leave the system in a 'tidy' state. A transaction is a series of invocations delimited by start-transaction and end-transaction markers. The effects of a transaction are deemed atomic if they appear as a single action. Thus, atomic transactions either successfully terminate, causing a permanent update or, if aborted part-way, leave no trace at all. A more detailed description of this scheme is given in [Dasgu86].

The first Clouds prototype, running on three bare VAX 11/750s, became operational in 1986. User access is through either discless SUNs or IBM PC-ATs, all machines being connected via Ethernet. The lengthy five year development period is attributed to Clouds not being built on UNIX or any other proprietary OS. Rather, it has been developed from scratch as a stand-alone operating system.

1.8.4 Cronus

The Cronus distributed operating system is under development at the BBN laboratories, Cambridge, Massachusetts [Schan86], [Gurwi86], [Schan87].

The primary objective of Cronus is :

> "To establish a comprehensive distributed system architecture and design for integrating a collection of different computer systems into a coherent, uniform computing facility that serves as a basis for developing distributed applications." [Schan86]

Cronus has three different classes of objects:

- ▷ Primal objects, forever bound to the processor that created them.

- ▷ Migratory objects that can move from processor to processor as situations and configurations change.

- ▷ Replicated and structured objects that have more internal structure than a single "atomic" object. An example is a reliable (replicated) file that has a number of identical primal files as its constituent parts.

Invocations upon these objects are performed by name only, without reference to an object's location.

Access control within Cronus is performed using *Access Control Lists* (ACLs). The goals of the access control mechanisms are :

> "To prevent unauthorized use of Cronus and Cronus objects; to preserve the integrity of the system; and to provide users with a uniform view of access control for all Cronus resources, services and objects." [Schan86]

An access control list is a list of permitted invokers for an object. Invokers not present on an object's ACL will not be serviced. Cronus extends this basic idea in two ways : by providing an ACL for each individual invocation routine; and by creating grouped object entries in an ACL rather than naming individual objects. Hence, an object belonging to an authorised group is permitted to make invocations, even though the object's name does not specifically appear in the control list.

The Cronus environment exists side by side with the original operating systems resident on the distributed hardware. This was a deliberate policy to ease the implementation effort, but also to allow users to gradually evolve to the new environment. Cronus currently runs on 15 hosts, including DEC VAXes running

VMS and UNIX, SUN workstations, BBNCC C70 UNIX systems, and several single-function Motorola 68000 microprocessor systems acting, for example, as file servers and authentication servers.

1.8.5 Emerald

Emerald is an object-based language and system designed at the University of Washington for the construction of distributed programs [Black86], [Jul 87]. Each Emerald object has four components :

- A name, which uniquely identifies the object within the network.
- A representation, which consists of the data stored in the object. The representation of a programmer defined object is composed of a collection of references to other objects.
- A set of Operations, which define the functions and procedures the object can execute. Some operations are exported and may be invoked by other objects, while others may be private to the object.
- An optional process, which operates in parallel with invocations on the object's operations. An object with a process has an active existence and executes independently of other objects. An object without a process is a passive data object and executes only as a result of invocations.

Emerald objects are fully mobile and can move from processor to processor within the network; even during an invocation. To support object migration the Emerald language includes a small number of location primitives. An object can :

- *Locate* another object, i.e., determine on which processor it resides.
- *Fix* another object at a particular processor.

- ▷ *Unfix* an object, i.e., make it movable following a fix.

- ▷ *Move* an object to another processor.

All object references are location independent. The system is responsible for mapping between an object's name and its current location.

A prototype Emerald kernel, running on top of Berkeley Unix, is currently operational. It runs on a small network of DEC MicroVAX II workstations connected by a ten megabit/s Ethernet. A prototype compiler has been constructed, capable of compiling simple Emerald objects into VAX machine code.

1.8.6 SOS

SOS is a general purpose distributed operating system, strongly influenced by the needs of office automation [Shapi86], [Makpa88]. It is a subtask of Esprit Project 367 "Secure Open Multimedia Integrated Workstation" (SOMIW). The goal of SOMIW is to construct an office workstation for manipulating, transporting and using multimedia documents that contain, for example, text, graphics, voice and moving images. The objective of the SOS (SOMIW Operating System) project is to design and implement a novel operating system based on the object oriented approach.

SOS is built around the *Proxy Principle* which states that :

> "In order to use some service, a potential client must first acquire a proxy for this service; the proxy is the only visible interface to the service." [Shapi86]

A proxy is a representative for one or more distributed objects that collectively provide a single service. The proxy is the only interface to the service and, to the

outside world, is indistinguishable from the service. Clients are unaware of the proxy's existence, believing they are invoking the service directly. Proxies, one per client, are always located at the same site as the clients they serve. This provides clients with a simple, local interface to (potentially) remote or distributed services.

A prototype version of SOS, operating on a set of workstations interconnected by a local area network, has been implemented on top of UNIX. The complete SOS network environment will ultimately consist of many local area networks interconnected by wide area networks. A bare-machine version is also in preparation.

1.9 Thesis

1.9.1 Background and Aims

The original aims of this research were to investigate the potential for distributed scheduling in an object oriented distributed environment. The intention was to identify attributes particular to object oriented systems that could be profitably exploited to improve upon current distributed scheduling techniques. These aims were later expanded to include wider resource provision issues, such as the automatic creation[1] of new object instances from class descriptions, and the 'enhancement' of existing objects to automatically endow them with properties such as fault tolerance. For each of the problems addressed, an object oriented solution was sought rather than applying existing, process based solutions. This thesis aims to demonstrate the benefits of exploiting the object paradigm for resource provision.

---

[1]for the moment read 'compilation'

1.9.2 Overview

Chapter 2 defines the Object Reference Model, which was developed to place the resource provision research into a well defined, conceptual framework. This model of object-oriented distributed systems is analogous in both spirit and design to the ISO seven layer model of communicating systems. It provides a logical framework relating the various aspects of distributed systems design, allowing different designs to be compared and contrasted. Chapter 3 describes the specific target environment addressed by later chapters. It states the assumptions made concerning distributed hardware, network capabilities, object invocation, object implementation, user population and envisaged applications.

Chapter 4 examines the problems posed by the need for multiple object representations in a heterogeneous environment. A distributed algorithm is developed for controlling the translation between different object representations, the primary use of which is to automatically create executable object instances from high level object descriptions.

An introduction to distributed scheduling is presented in chapter 5. In particular, the special requirements for scheduling in an object oriented environment are discussed. Based on these requirements chapter 6 develops a scheduling mechanism, known as comparison scheduling, that operates by applying statistical hypothesis testing techniques to object oriented performance metrics. Simulated results of the scheduler's performance are compared to intuitive 'control' schedulers. Chapter 7 applies the statistical techniques employed in comparison scheduling to the suppression of redundant status update messages. Simulation results are presented, suggesting that considerable reductions in update traffic can be realised without degrading the scheduler's performance.

Chapter 8 introduces the concept of virtual objects as a paradigm for structuring distributed systems. Virtual templates are presented as a general mechanism

for creating virtual objects, by encapsulating useful properties such as resilience and persistence in a re-usable form.

The various aspects of resource provision are drawn together in chapter 9, which speculates on how distributed resources might be provided to users. The techniques developed in this thesis are then presented as a solution to users' resource provision needs. The effects upon resource provision of user actions and system configurations are also discussed. Finally, chapter 10 provides a summary of the thesis, examining the possibilities for future work and presenting the conclusions reached.

Chapter 2

The Object Reference Model

This chapter defines the Object Reference Model (ORM). The background to the model is examined, indicating the assumptions made concerning distributed hardware, network facilities, applications and users. A list of design problems to be encapsulated by the model is then developed. The ISO Open Systems Interconnection model is discussed, in particular, the OSI layering principles. These layering principles are applied to the design problems identified earlier, to yield the ORM layers. Each layer's purpose is then defined and examined in detail, with illustrative examples taken from the systems introduced in chapter 1.

2.1 Introduction

The Object Reference Model (ORM) is a conceptual model providing a framework for the development of object oriented distributed systems. In particular, it provides a framework for describing the research in this thesis.

ORM identifies the generic design problems associated with distributed systems, modelling the interrelationships between them. It does not specify particular services or protocols, rather, it identifies the general nature of the services and protocols required to construct an object oriented distributed system. The functional breakdown it provides can also be used to guide the decomposition of existing designs, thereby assisting in the analysis of current systems.

Producing an abstract model to aid understanding in a particular area of systems design has proved useful in the past. Perhaps the best known example is the International Standards Organization (ISO) seven layer reference model of Open Systems Interconnection (OSI) [ISO 81]. This model of computer communication systems, examined below in section 2.6.1, provided the original inspiration for ORM's development. The International Standards Organization is currently developing a model of Open Distributed Processing (ODP). The reference model for ODP [ISO 89] is based upon the five **Viewpoints** (Enterprise, Information, Computation, Engineering and Technology) developed by the ANSA (Advanced Networked Systems Architecture) project [ANSA 89].

The purpose of the **Enterprise Viewpoint** of ANSA is to provide a framework for explaining and justifying the role of an information processing system within an organization. An enterprise model describes the overall objectives of a system in terms of roles (for people), actions, goals and policies. It specifies the activities that take place within the organization, the roles that people play in the organization, and the interactions between them.

The **Information Viewpoint** provides a framework for describing the information requirements of a system. An information description of a system is made up of information elements, rules stating the relationships between information elements, and constraints on the information elements and rules. Information models must also show both how information is partitioned across logical boundaries, and the required quality attributes. Information models do not have to differentiate between parts that are to be automated and parts that are to be performed manually.

The **Computation Viewpoint** provides a framework for modelling the operations of information transfer, retrieval, transformation and management necessary to automate information processing. The mechanisms required to support a computation model are specified in the engineering viewpoint of the system. A computation model of a system partitions the required transformations among processing objects as necessary to achieve the complete set of transformations. The partitioning thus defined is logical and not location-dependent. A computation describes the structuring of applications independently of the computer systems and networks on which they run.

The **Engineering Viewpoint** provides a framework for describing how to mechanize the concepts identified in the computation model. This will include a definition of the physical distribution (as required) to realize the partitioning defined in the computation projection.

The **Technology Viewpoint** provides a framework for describing the technical artifacts (realized components) from which the distributed system is built. It shows how the hardware and software that comprise the local operating systems, the input/output devices, storage, and points of access to communications, are mapped onto the mechanisms identified in the engineering model.

The ODP reference model, based around these viewpoints, provides a framework for describing different aspects of distributed systems. The work developed

later in this thesis lead to the development of ORM to reflect the computation aspects of resource provision. Therefore, the Object Reference Model defined in this chapter lies firmly within the computation viewpoint, and, using ANSA terminology, is therefore a computation model. The examples of real systems used throughout this chapter, and the resource provision mechanisms developed throughout the remainder of this thesis, lie within the engineering viewpoint.

A model addressing similar issues to ORM is presented by Watson in [Watso81]. This model identifies the generic problems associated with distributed system design, incorporating them into a series of hierarchical layers. Although Watson's model uses the term 'object', it is not object-oriented in the sense defined by

chapter 1. It also concentrates heavily on the communication aspects of distributed systems, an area that has subsequently received considerable attention.

ORM has been developed along the same lines as Watson's model. However, ORM specifically addresses object oriented systems. Another major difference is in its treatment of communication. ORM assumes a comprehensive underlying communication facility is readily available, and therefore concentrates on the more abstract problems posed by distributed systems.

2.2 Background to the Model

Before defining ORM it is necessary to establish which aspects of object oriented distributed systems are to be modelled. Figure 2–1 shows the general structure of object oriented distributed systems from an individual user's perspective. The Object Reference Model is concerned only with the **Object Environment** component of the figure. However, before examining the Object Environment component in more detail, the following sections examine the components above and below, stating any assumptions made by ORM.

2.2.1 Distributed Processors

ORM assumes the processors in a distributed environment are heterogeneous[1]. The model does not restrict the size, type or number of processors. The only requirement is that each processor is capable of supporting an object oriented interface, i.e., capable of receiving, interpreting and responding to invocation messages.

---

[1]A homogeneous system is a special (simple) case of heterogeneity.

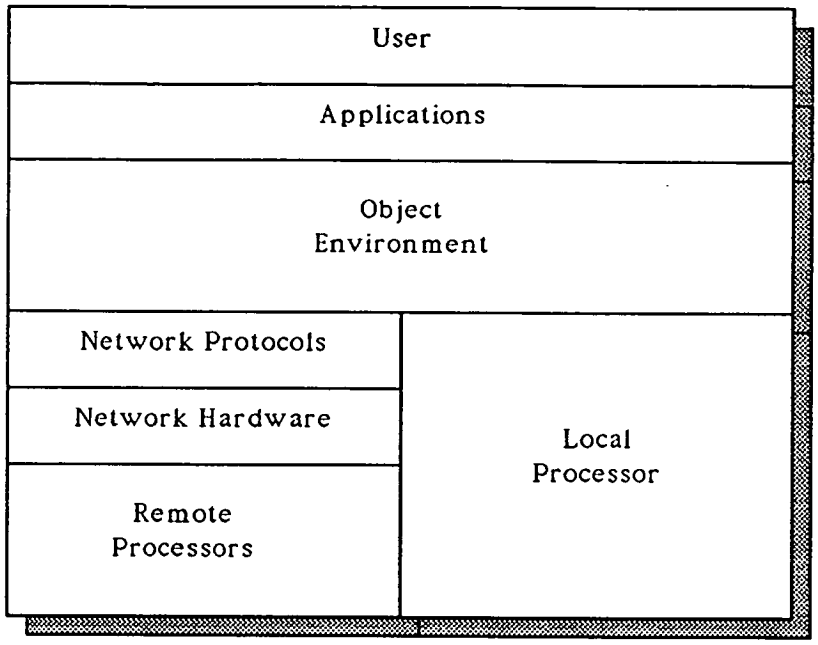

Figure 2–1: General Structure of Object Oriented Distributed Systems

This requirement is readily met by general purpose processors such as workstations. The problem is simply one of creating a suitable software environment, for example, an object oriented operating system. This is a task that, by their very nature, general purpose processors are well suited to. Specialist processors on the other hand are generally less adaptable, being designed exclusively to perform one task well. In cases such as this, a 'front-end' processor can be employed. A front-end processor is a general purpose processor that nominates to provide an object oriented interface on behalf of a specialist processor. Invocations are directed towards the front-end, which mediates the non-object oriented access to the specialist processor. Invokers, unaware of how the service is provided, perceive a service accessed in the usual manner, i.e., via an object oriented interface.

Hence, either directly or by employing front-ends, all processors within a distributed environment can support an object oriented interface.

2.2.2 Network Protocols

The network protocols are assumed to provide transmission facilities for a set of simple types such as integers, reals and characters, including any compound structures composed from these. The network is also expected to provide automatic translation of these types, masking any differences in the representation used by heterogeneous processors. It is further assumed that facilities exist to support encryption of invocation messages during transmission.

As an example, the ISO standard *Abstract Syntax Notation One* [ASN.1] addresses some of these issues. It defines, at the bit level, the representation to be used when transmitting a variety of simple data types such as booleans, integers and character strings. Provision is also made for the encoding of sequences and sets, allowing structures such as arrays and records to be transmitted as complete units.

ORM assumes no particular network configuration. For example, both wide area and local area networks are encompassed, subject to provision of the above services by the protocols used. However, as indicated by those examined in chapter 1, current systems exclusively employ local area networks as the primary transmission medium. This preference is a result of the superior performance characteristics of local area networks. LANs typically provide fast transmission over a limited distance, with relatively few errors. In contrast, wide area networks typically transmit over unlimited distances, but with less speed and a greater error rate. Wide Area Networks are therefore used, if at all, only to provide limited inter-operability between geographically dispersed LANs.

2.2.3 Applications and Users

Applications use the facilities provided by the Object Environment to perform tasks on behalf of users. ORM places no restrictions on the type or number of

applications, or on the number of users. In any particular system, constraints are likely to be defined by the type and number of processors available, and by the network capabilities.

2.3 The Object Environment

The Object Environment provides the link between the resource requests of applications and the resources provided by the distributed environment. It takes a collection of heterogeneous processors, presenting them as a single, coherent system that supports the object oriented paradigm. In general, the Object Environment is responsible for extracting and controlling the benefits of distribution and object orientation, presenting them in a uniform manner.

Uniformity can be created on several levels. First, uniformity of presentation. All resources, regardless of their nature, are presented as objects. No other form of resource exists. Second, uniformity of naming. Using location independent, global naming schemes an object is referenced by the same name, regardless of where it is located. Finally, uniformity of access. The *only* method of accessing a resource is via an invocation message.

The Object Environment is responsible for providing and maintaining this uniformity. The following section examines in detail the design problems associated with this.

2.4 Model Requirements

ORM models the distributed operating system architectures used to realise the Object Environment component of Figure 2–1. This section defines the key design issues to be included within ORM. Each was adopted after careful examination of many current designs, including those described in section 1.8. Collectively these issues summarise the broad spectrum of design problems faced by distributed system designers :

- ▷ Object naming and addressing
- ▷ Object mobility
- ▷ Object representation and construction
- ▷ Object scheduling
- ▷ Invocation scheduling
- ▷ Security and protection
- ▷ Reliability
- ▷ Consistency

2.4.1 Object Naming and Addressing

The Object Environment presents applications with a global address space, populated exclusively by objects. It must therefore mask location dependencies, building upon a distributed, network name-space containing processor names, to create

a unified object name-space containing only object names. The Object Environment is therefore responsible for mapping between object names and network addresses. The naming issues involved pervade all levels of system design.

2.4.2 Object Mobility

As indicated earlier, the Object Reference Model concentrates on the more abstract features of distribution rather than on communication. The lowest level of communication problem addressed by ORM is Object Mobility, i.e., migrating object representations between processors. Although it is assumed the underlying communication system provides a data transmission facility, the minimum communication requirements stated in 2.2.2 are somewhat limited in scope. Many systems may require more advanced facilities, for example, communicating abstract data types as 'atomic' structures. Although some networks may provide these advanced facilities, they are not universal. The Object Environment must therefore incorporate a communications element to cover such cases.

2.4.3 Object Representation and Construction

Every object exists in at least two forms : its human readable representation (usually textual); and its machine representation (usually binary). Several intermediate representations are normally required to transform between the two. These can be thought of as functionally equivalent representations of the same object, at different levels of abstraction. Multiple representations may also exist at the 'same' level of abstraction. For example multiple human representations corresponding to different implementations, possibly in different languages. Similarly, multiple machine representations may also exist, particularly in a heterogeneous environment where each different type of processor requires a different binary format.

In most current systems, users explicitly manipulate multiple object abstractions. For example, selecting the compiler to be used, passing it the class descriptions ('source' code) to produce an executable representation that can be run. In some cases, more than one step may be required to translate from class descriptions to executables, involving, for example, assemblers and linkers. All of these steps are reliant upon user intervention.

Whilst encompassing this approach to object construction, the Object Reference Model includes provision for automatic generation of executable instances from class descriptions. The motivation for this is to allow users to work exclusively with high-level object representations; all other representations, and the transformation mechanisms employed to create them, being controlled by the system. This is similar to the approach taken by the non-distributed object oriented environment SmallTalk-80 [Goldb83].

2.4.4 Distributed Scheduling

Location independent naming gives the Object Environment freedom to locate objects as it sees fit. Many systems therefore attempt to make assignments that optimise resource usage. The two principal components of distributed scheduling are the assignment of object instances to processors and the assignment of invocation messages to object instances.

Object Scheduling

Object scheduling is the assignment of object instances to processors. Judicious placement of object instances can assist in optimising resource usage. When a new instance is to be scheduled, the scheduler observes the status of each processor and selects the 'best' one (according to some criteria) to host the object. This form of

object scheduling is known as **Load Sharing**, because the load generated by new object instances is shared between the processors.

Some systems also support **Load Balancing**. Load balancing is a particular form of load sharing where the objective is to continually maintain an equal workload on each processor. This usually involves migrating previously scheduled objects in mid-execution to compensate for transient fluctuations in load distribution. When an object migrates, the system must ensure that subsequent invocations are automatically bound to the new location (see Invocation Scheduling below).

Load balancing is more expensive than load sharing, both to implement and perform, because of the complexity introduced by the need to transfer an object's execution state. This complexity becomes even greater in a heterogeneous environment where the representations used by the originating and receiving processors may be different. The Object Reference Model must include provision for both load sharing and load balancing.

Invocation Scheduling

Invocation scheduling involves binding invocation messages to object instances. In a distributed environment an object name may bind to several different instances, each one capable of servicing the invocation. Consequently, some form of scheduling mechanism must be employed to select the recipient object. In such cases, optimisation of resource usage can be assisted by directing invocations to the 'best' instance (according to some criteria).

2.4.5 Security and Protection

Security and protection issues are concerned with governing access to resources. In a system constructed entirely from objects this principally involves verification of access rights when objects are invoked. The need for access control arises from

the desire to share resources (see section 1.3.4). If users only had exclusive access to their own resources, isolated from all others, then access controls would be largely redundant.

Protection is required against malicious attempts to use or abuse restricted resources; for example, by the impersonation of authorised users. Accidental misuse caused, for example, by the incorrect implementation of an invoking object instance must also be protected against. The security mechanisms employed in a distributed system must therefore enable the recipient of an invocation message to verify the identity of the message sender. The recipient must also be confident that the contents of the received message are exactly as the sender intended. ORM must model the many different security mechanisms employed in current systems.

2.4.6 Reliability

The service provided by an object instance is subject to the reliability of the processor on which it resides. Within a distributed environment, occasional failures can be masked by utilising redundant hardware, software and data. ORM must allow for the provision of reliability, as well as any other related aspects of fault-tolerance such as persistence and availability.

2.4.7 Consistency

In a distributed environment an object may be replicated many times. There are several reasons why this may be useful; for example, to improve reliability and availability, or simply to offer a 'larger' service than can be provided by a single instance. Uncoordinated invocations on individual instances of a replicated object, leads to inconsistencies between the instances' states. The result of an invocation becomes dependent upon the particular instance invoked. In some cases this may be perfectly acceptable. However, if the instances logically provide a single, global

service, then they should be synchronized to present a coherent global state. The Object Reference Model must include provision for consistency protocols.

2.5 Virtual Properties

Within ORM, reliability, persistence, consistency and access controls are collected together as examples of **Virtual Properties**. Typically, virtual properties either mask inherent limitations of the underlying system (e.g., equipment failure) or enhance an object's interface to match that expected by its invokers (e.g., access control mechanisms). Virtual properties are therefore characteristics possessed by objects that somehow 'improve' the service offered, but in a service independent manner.

Section 2.7.5 identifies several virtual properties, using the example systems to illustrate details of their implementation. In current systems, virtual properties are usually coded explicitly within an object's implementation. Although ORM does not deny this mode of implementation, it models virtual properties at their logical level of abstraction, i.e., as being added 'on-top-of' existing services, rather than being 'built-in'.

Having identified the design aspects to be incorporated in the Object Reference Model, the following section examines how such a model can be derived.

2.6 Layering

ORM is a layered model. This section examines the principles behind layering, using the ISO Open Systems Interconnection model as an example. The ISO layering principles are applied to the design problems defined above to produce the Object Reference Model layers.

According to the layering technique, each system is viewed as being logically composed of an ordered set of sub-systems represented as a vertical sequence of layers, each layer encapsulating a particular aspect of system design. Adjacent layers communicate through their common interface, each layer building upon and adding value to the services of layer below it. The topmost layer presents a service incorporating those provided by all lower layers.

Modular and layered design is widely accepted as good software engineering practice. The following list identifies some of the reasons for this :

- The internal structures, mechanisms, encodings and algorithms used within a layer are not visible to other layers, i.e., each layer is encapsulated.
- Complex systems can be decomposed into more easily understood pieces.
- The implementation of a given layer can be changed without affecting the service offered, provided the layer interface remains the same.
- Alternate implementations for a layer can coexist.
- A layer can be simplified or omitted when any of its services are not needed.
- Confidence in the correct operation of a layered system is more easily established by testing and analysis of each layer in turn.

- ▷ Precise specification of layer functions, services and interfaces encourages standardisation.

These characteristics are very similar to those defined in chapter 1 for object orientation. Layering is sometimes accused of yielding inefficient implementations. This criticism would apply to ORM if it were used as a literal template for implementation. However, the ORM layers model the *logical* relationship between the various design aspects modelled. They are not intended to provide a literal template for an implementation, which, for efficiency, may combine the logical functions of multiple layers in a single 'program'.

The design of a suitable model involves identifying layers that adequately describe the problem. This is a somewhat arbitrary choice, influenced by the modeller's point of view. Many such layers could be envisaged and it is impossible to prove that any one design is *best*. It is only possible to state that one design is better than another when measured against some recognised criteria.

The following sections describe the ISO Open Systems Interconnection model. This discussion serves several purposes. Primarily it introduces the layering design rules used to develop the OSI model. These rules form the basis upon which the ORM layers are defined. Second, it provides an example of a widely used layered model. Finally, it describes the type of communication facilities ORM assumes to be provided by the underlying network. ORM can in fact be thought of as extending the OSI model to incorporate object oriented distributed systems.

2.6.1 The OSI Layering Principles

The Open Systems Interconnection (OSI) initiative grew from the need to create internationally agreed, standardised communication protocols. In order to assist such a large undertaking the International Standards Organisation (ISO) defined a communications reference model. The function of this model was not to specify

any particular protocol, but to provide a framework within which new protocols could be developed. It further provided a means of categorising existing protocols.

In order to assist the modelling process, and help justify the resultant design, a set of layering principles were devised [ISO 81] :

1. Do not create so many layers as to make difficult the system engineering task of describing and integrating these layers.

2. Create a boundary at a point where the description of services can be small and the number of interactions across the boundary are minimised.

3. Create separate layers to handle functions that are manifestly different in the process performed or the technology involved.

4. Collect similar functions in the same layer.

5. Select boundaries at points that past experience has demonstrated to be successful.

6. Create a layer of easily localised functions so that the layer could be totally redesigned, and its protocols changed in a major way to take advantage of new advances in architectural, hardware or software technology, without changing the services and interfaces with adjacent layers.

7. Create a boundary where it may be useful at some point in time to have the corresponding interface standardised.

8. Create a layer when there is a need for a different level of abstraction in the handling of data (e.g., morphology, syntax, semantics).

9. Enable changes of functions or protocols within a layer, without affecting the other layers.

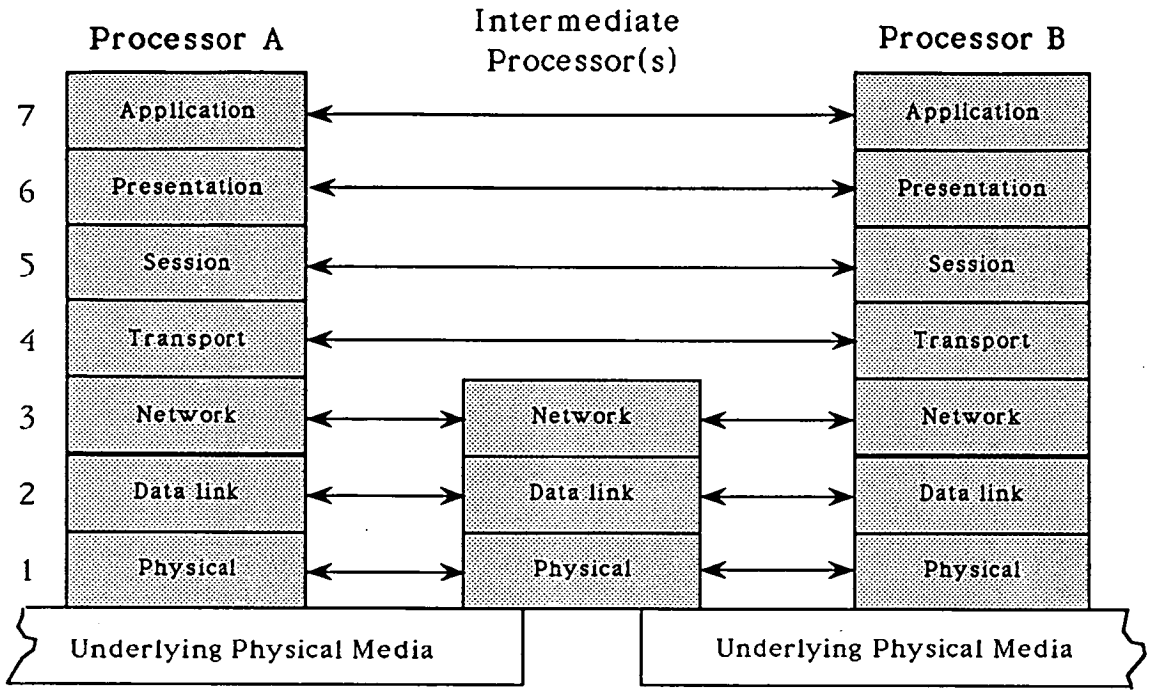

Figure 2–2: OSI Reference Model Layers

10. Create for each layer, interfaces with its adjacent layers only.

These principles can be summarised as embodying the well established software engineering techniques of problem partitioning, information hiding and minimising interfaces.

2.6.2 The OSI Layers

The layers defined by the OSI model are shown in Figure 2–2. The arrows show peer-to-peer communication between layers.

The **Physical Layer** is concerned with unstructured bit transmission between point-to-point links. Layer 1 protocols specify the mechanical and electrical properties of transmission media and their access protocols.

The **Data Link Layer** provides the functional and procedural means to activate, maintain and deactivate one or more physical links. Layer 2 protocols may allow reliable transfer of data across physical links.

The **Network Layer** is responsible for transferring messages from source to destination over an arbitrary succession of data links. It provides the upper layers with independence from the data transmission and switching technologies used to connect systems.

The **Transport Layer** is concerned with end-to-end data transfer between session entities. Layer 4 protocols hide transmission methods from the upper layers.

The **Session Layer** establishes, maintains and terminates connections between applications.

The **Presentation Layer** provides a data translation facility to hide the syntax of transmitted data from applications. Layer 6 protocols specify data formats that enable cognizant exchange of information between dissimilar machines.

The **Application Layer** provides application programs with access to the OSI environment.

It can be seen that each layer enhances the services of the layer below, and is a pre-requisite for the layer above. The higher the layer the greater the level of abstraction. In principle, layer 7 provides an error free, end-to-end link between any two world-wide sites, independent of transmission media, switching technology, data speed, data representation, and so on.

The ISO model of Open Systems Interconnection is not universally accepted as suitable for describing all communicating systems. OSI was originally designed with wide area networks in mind. As such, the layers are strongly influenced by the assumption of point-to-point communications — where a connection is opened,

with both sender and receiver establishing connection related status information; (large quantities of) data are passed across the opened connection; and the connection is then closed — rather than the datagram, or broadcast, modes used in most local area networks — where each datum is a self contained 'package' that is sent independently of any others. Consequently, the OSI model is often overly elaborate for describing the communication facilities of distributed systems based on local area networks. These arguments do not really concern us here, as the main purpose of discussing the OSI model was to introduce the layering principles, which are largely uncontroversial.

2.6.3 Applying the OSI Principles to ORM

The OSI layering principles are not specific to communication models. They provide general guidelines for identifying layers and can therefore be used to assist the development of a layered model for any design problem. The design problems addressed by ORM were defined earlier in the chapter. Although subjective, they are based upon careful examination of many distributed system designs. The OSI layering principles (page 46) can be applied to these design problems to produce a layered model of object oriented distributed systems.

The following paragraphs give a brief explanation of how the ORM layers were chosen. Principles 1, 2, 9 and 10 are general directives that apply to all layers. Although specific layering principles are mentioned below in relation to each layer, these are identified only as being *more* applicable than the others. More important than any one specific principle is the *spirit* they collectively convey. In general therefore, all 10 principles apply to all layers.

The communication services specified by the OSI model, and those typically provided by local area networks, do not directly support object oriented interactions. Application of principles 3, 5 and 8 therefore leads to the identification of a

Migration Layer as the lowest layer in the Object Reference Model. The Migration Layer builds upon the available network services to provide an object oriented communication service, migrating objects and invocation messages between locations in the distributed environment.

The inherent heterogeneity within a distributed environment calls for some control over multiple object representations. Application of principles 3, 7 and 8 yield the **Construction Layer**. The Construction Layer services provide transformation and translation facilities for constructing and manipulating the multiple object representations required in a heterogeneous environment.

Scheduling mechanisms are required to control the assignment of object instances to distributed locations. Application of principle 6 leads to the **Location Layer**, whose services provide location-independent object scheduling. Applying principle 6 again yields the **Invocation Layer**, which embodies a location-independent invocation service.

The services represented by the lower four layers allow for the creation and invocation of object instances. Incorporating virtual properties, with the application of principles 4, 6 and 7, leads to the **Virtual Layer**. The Virtual Layer services encapsulate techniques for manipulating object instances and invocations to provide virtual properties.

Finally, the services offered by these layers must be made accessible to users. The **User Layer**, the uppermost ORM layer, presents these services in a form suitable for users and their applications.

The ORM layers, which are examined in detail below, are shown in Figure 2–3. This particular diagram shows a configuration consisting of four processors. The lower three layers — Migration, Construction and Location — operate in location-dependent address spaces, and therefore appear 'stacked' in the same manner as the OSI layers. However, the upper layers — Invocation, Virtual and User

— operate in location-free address spaces. They therefore appear as continuous bands, independent of any particular location.

2.7 The ORM Layers

This section examines the ORM layers in detail. For each layer, the naming and addressing issues are examined, followed by a description of the services offered. Examples, taken from the systems described in chapter 1, are used for further illustration.

At the highest ORM layer, the User Layer, only virtual objects exist, with no concept of location or distribution. This address space, containing only location-free, virtual objects, is created successively by each layer. Layer N operates within the address created for it by layer $N-1$, refining it to present a more abstract address space to layer $N+1$. Hence, the address space provided by the underlying communication services is built upon to provide the location free address space seen by users.

The services embodied by each layer are divided into operational and management activities (Figure 2–4). The operational services of layer N are the standard services offered to layer $N+1$. There are two main operational themes common to each layer : object-provision; and object-invocation. The object-provision services support the creation and deletion of object instances. The object invocation services provide access to these objects. The management services of layer N also have two themes : internal-management; and external-management. The internal-management services give layer N the management information and controls necessary to provide the layer N service. The external-management services are those offered to layer $N+1$, enabling it to control the application of, and perhaps customise, the layer N operational services.

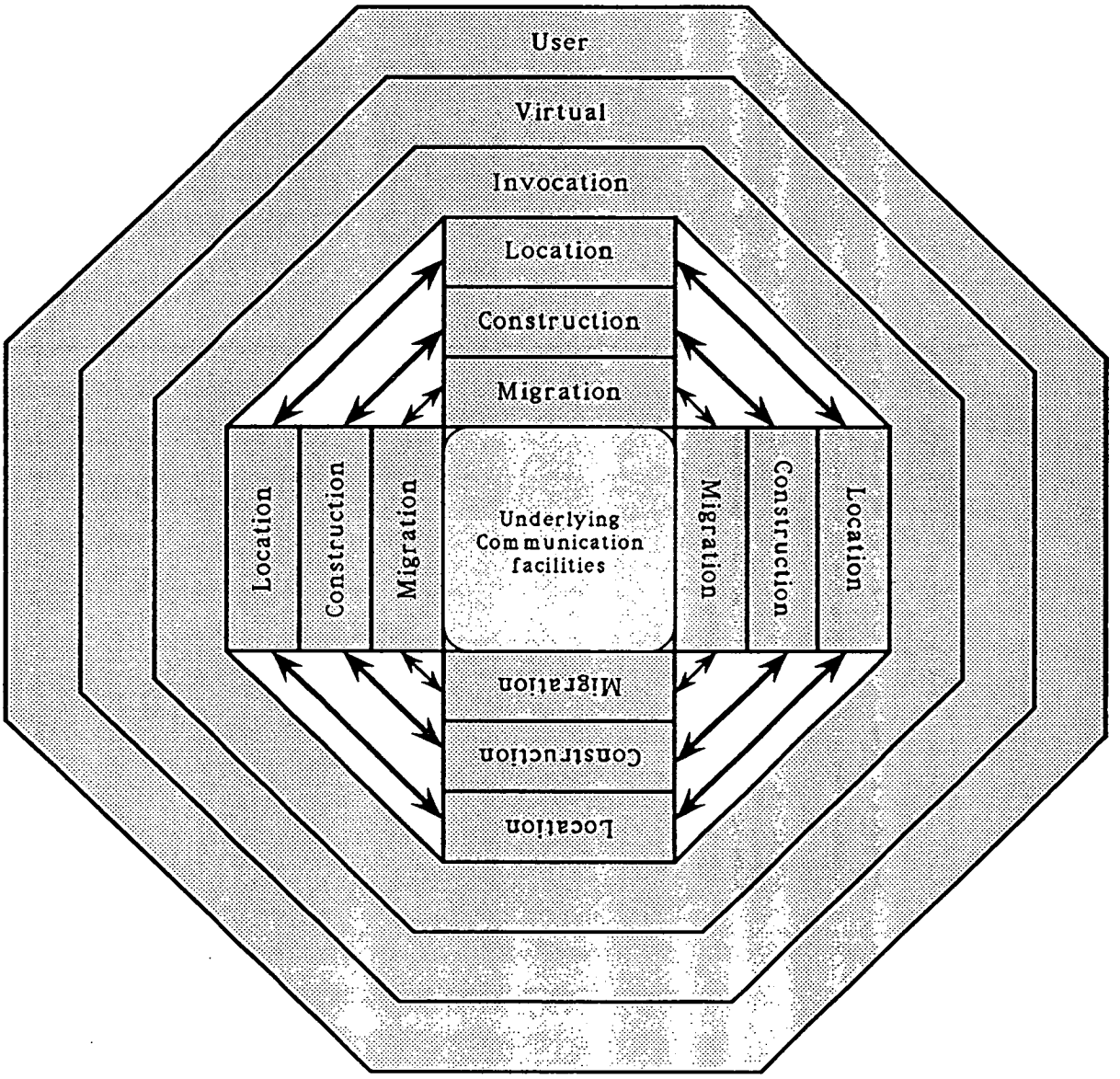

Figure 2–3: The Object Reference Model Layers

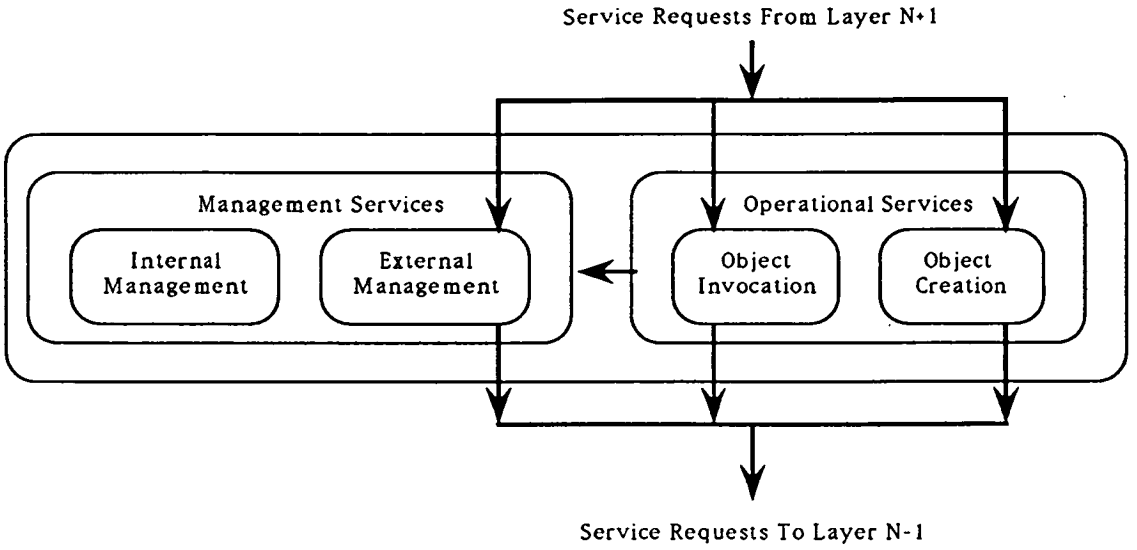

Figure 2–4: Layer N Operational and Management Services

2.7.1 The Migration Layer

The Migration Layer handles the communication of object representations between different locations in the system, building upon the facilities provided by the underlying network.

Naming and Addressing

Within the Migration Layer, location addresses correspond to those provided by the underlying **Communication Address Space**, i.e., the addresses of processors. Object names refer to specific object instances residing at specific locations, for example, `ObjectRepresentation@ProcessorAddress`.

The Migration Layer maps names and addresses from the **Construction Address Space** used by the Construction Layer onto the *communication address space.* Names in the *construction address space* do not refer to specific processors; rather, they identify 'hosts' at which objects can be placed. A host may in fact correspond to a particular processor or, alternatively, several 'logical' hosts may map to one

processor (chapter 3 provides a more detailed definition specific to the environment addressed by the remainder of the thesis). These relationships are hidden by the Migration Layer. Names and addresses in the *construction address space* therefore remain location dependent, but are no longer network dependent, for example, `ObjectRepresentation@Host`.

Operational Services

The Migration Layer services provide the Construction Layer with object migration facilities in which the fundamental unit of communication is an object. It builds upon the facilities offered by the underlying communication service, to provide the object based migration facilities expected by the higher layers. An example migration layer service for object-provision is an operational service such as :

```
Move(ObjectRepresentation, FromHost, ToHost, Options)
```

The purpose of the `Options` parameter(s) will vary from system to system. It provides any additional information required to perform the `Move` operation. For example, it may specify that encryption should be used, or that this is a high priority request and hence the fastest available transmission service should be used. System specific information such as 'hints' are also modelled by the options parameters. As an example, a system 'hint' might suggest how the service should be provided. For completeness and generality, a similar `Options` parameter is added to all example services throughout this section.

As indicated earlier, the services provided by local area networks do not always conform to those identified in the OSI reference model. The 'size' of the Migration Layer will therefore vary between systems, depending upon the level of services offered by the underlying network. In systems built upon advanced communication facilities, some of the migration services will map directly onto those

of the network. In systems where the network provides minimal functionality, the Migration Layer services must perform more work in order to create the required facilities.

The Migration Layer services are responsible for performing any high level data transformations associated with transmission, such as mapping the data structures used to represent objects onto the data transmission primitives offered by the network. However, these services are not responsible for type matching executable representations with processors, which is a function of the Construction Layer services.

The Migration Layer's contribution towards object-invocation is the migration of invocation messages between objects. This is essentially the same as the object-provision service, since invocation messages are themselves objects. Similar `Options` are therefore possible, for example, placing priorities on message delivery, or ensuring message integrity by encryption.

Management Services

External-management services are offered to the Construction Layer to assist in its use of the migration facilities. These services should provide information relating to the 'availability' of hosts within the *construction address space*, reflecting system failures such as network partitions and processor crashes. For example :

```
IsAvailable?(ThisHost)
```

might inform the Construction Layer whether a particular host is reachable. The internal-management services assist the Migration Layer in using the underlying communication facilities. For example, monitoring communication performance, maintaining configuration information, and adapting to new or changed facilities.

Examples

In general, current systems perform object migration at the binary level, with no attempt made to recognise any higher level structures (structured object migration is examined later in chapter 5). This implies that object migration can be implemented directly by the underlying data transmission facilities. The following examples therefore concentrate on invocation services rather than migration services.

In many distributed systems, object invocation is performed using **Remote Procedure Calls** (RPC). RPC extends the well understood procedure call mechanism into a distributed environment [Birre84]. When a remote object is invoked, the calling environment is suspended. The invocation message is then constructed and passed across the network to the remote object, where the desired invocation routine is run (Figure 2–5). When the invocation routine is completed, the results are passed back to the calling environment, where execution resumes as if returning from a simple, single-machine call. Both the invoking object and the invoked object, known as the **Client** and **Server** respectively, remain unaware of the RPC mechanism, each observing only a local invocation.

In some systems the basic RPC facilities may be provided by the underlying communication facilities. Under these circumstances the Migration Layer services must map invocations onto the appropriate RPC routines. If no RPC facilities are available, then the Migration Layer services must build upon the communication facilities to provide a complete invocation mechanism.

The SOS system provides an extensible invocation service constructed progressively upon the underlying communication service through the use of inheritance [Makpa88]. A basic host-to-host protocol, providing facilities common to all invocation protocols, is encapsulated within a `BaseProtocol` object. An invocation protocol is defined by two objects; the `ProtocolObject` and the

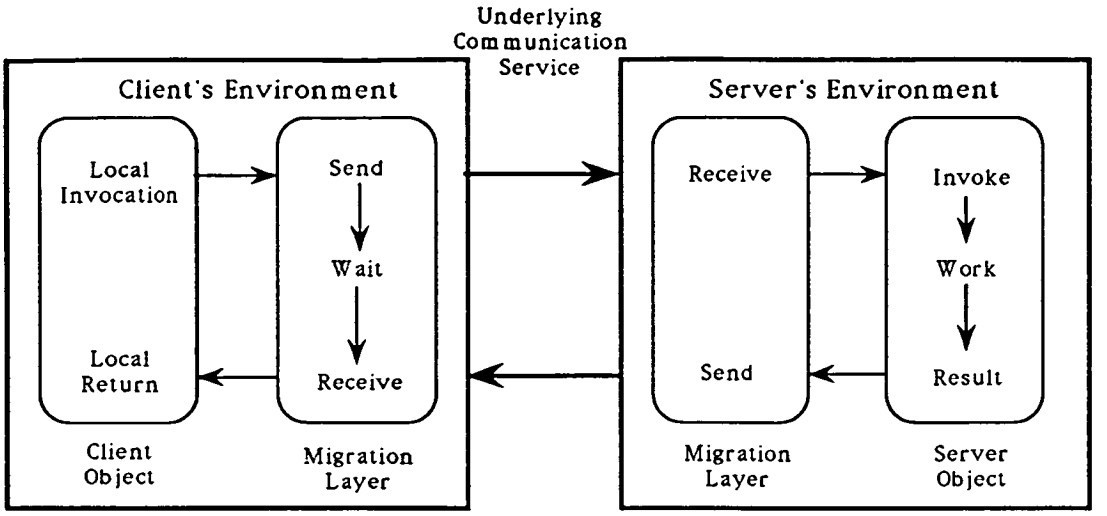

Figure 2–5: A simple Remote Procedure Call

`ProtocolManager` object. The protocol object encapsulates the services specific to the protocol it implements, inheriting the common facilities from the basic protocol object. The protocol manager object has three functions :

- ▷ Establishing connections. When asked, the protocol manager establishes a communication link between objects.
- ▷ Managing protocol resources. Each connection has a descriptor in a connection table identifying the caller, the callee and the address of the local protocol object for this connection.
- ▷ Dispatching invocation messages. It is the role of the protocol manager to forward invocation messages to the appropriate protocol object for transmission.

There is one protocol manager per protocol type, per processor. There are two protocol objects per connection, one at each end of the link.

The object oriented design of SOS's communication facilities enables the progressive development of invocation protocols. New protocols are built upon exist-

ing protocols utilising the object oriented inheritance mechanism, thereby encouraging extensive re-use of code and design.

2.7.2 The Construction Layer

The Construction Layer services provide transformation facilities for manipulating object representations. They encapsulate the inherent heterogeneity of the underlying environment, masking it from the higher layers.

Naming and Addressing

The Construction Layer operates within the *construction address space* defined earlier. It encapsulates the techniques used to manipulate individual object representations at different levels of abstraction. The complexity of multiple representations is hidden from the Location Layer by creation of the **Location Address Space**. Within this address space the concept of location remains the same, i.e., a repository for objects. However, explicit references to different representations of the same object are replaced by a generic object name, for example, `ObjectName@Host`. It is the Construction Layer's responsibility to map between the *location address space* and the *construction address space.*

Operational Services

The principal object-provision service embodied by the Construction Layer, is the generation of object representations for execution at specified locations within the *location address space.* In order to perform this task the Construction Layer services must be aware of all object representations used within the system. They must also be aware of the translation tools available to manipulate these representations. For example, a construction service must be able to associate a host

with a particular representation, and associate the representation with the transformation tools used to generate it. Construction rules are required to control the application of the transformation tools to generate the desired representation.

The Construction Layer therefore provides the Location Layer with operational services such as :

```
MakeInstance(ObjectName, @Host, Options)
```

The object name belongs to the *location address space.* The Construction Layer services must identify the host's type, mapping the object name onto the appropriate (representation dependent) *construction address space* name before examining appropriate transformations. An example use of the options parameters may be to specify exactly how the instance should be produced, overriding any default or automatic mechanisms.

An object-invocation service should also be provided for the forwarding of invocation messages. For example :

```
Invoke(ObjectName, @Host, InvocationMessage, Options)
```

If necessary, the invocation message and its parameters can be transformed into the representation expected by the recipient object.

Management Services

The external-management services offered to the Location Layer provide information on possible transformations. For example, the boolean service :

```
CanLocate?(ObjectName, @ThisHost)
```

might indicate whether a suitable representation of `ObjectName` can be created to execute at location `ThisHost`. Note that no information is returned as to *how*

the representation would be generated, it simply indicates the feasibility. Other possibilities include providing information on the cost of transformations. For example, the function :

```
WhatCost?(ObjectName, @ThisHost)
```

could provide some measure of the resource costs involved in creating the appropriate representation, enabling different possibilities to be compared.

The internal-management issues relate to the actual provision of a construction service. In many current systems, most representation manipulation is performed explicitly by users. In this case, the internal-management is trivial and amounts simply to providing users with access to the transformation tools (compilers, linkers and so forth). In a more sophisticated environment where the system takes over some, or all, of the transformation responsibilities, there will be a need to maintain a database of known transformation tools, along with rules for applying them.

Examples

Generally, in current systems, including those mentioned in chapter 1, users are entirely responsible for object construction. They are aware of and directly manipulate different object representations, for example, by invoking compilers and linkers. If representations are required for more than one processor type then users generate these explicitly. The systems therefore only 'understand' object representations at the binary level. All higher level abstractions remain meaningless, requiring user intervention before they can be interpreted.

An example of parameter transformation is provided by a `printDocument` invocation routine for printer objects. The text of the document object to be printed (e.g., ASCII), which is passed as a parameter, may require translation into a form

accepted by the printer (e.g., PostScript). Many similar examples of parameter translation could be envisaged.

2.7.3 The Location Layer

The *location address space* in which the Location Layer operates contains only objects and object repositories. It is the function of this layer to control the assignment of objects to repositories. It encapsulates the location dependencies inherent in a distributed environment, hiding them from the higher layers.

Naming and Addressing

The Location Layer services hide location dependencies from the invocation layer by the presenting a location-free **Invocation Address Space**. Only objects exist in the *invocation address space*, with no concept of location. For example, the *invocation address space* name `ThisObject` may map to the *location address space* name `ObjectName@Host`.

Operational Services

The basic object-provision services embodied by the Location Layer are the scheduling and de-scheduling of object instances. For example :

```
Schedule(ThisObject, Options)
DeSchedule(ThisObject, Options)
```

In order to schedule an object the scheduler uses the status information provided by its internal-management services (see below). This information is assessed, perhaps in conjunction with the external-management information provided by the Construction Layer, to decide which location should host the new

instance. The criteria against which this decision is made are defined by scheduling policy. Once a location has been selected, the decision is passed to the object-provision services of the construction layer, which then creates the appropriate executable instance. In the case of a de-schedule operation, the Location Layer simply deletes the appropriate instance.

The Location Layer plays no part in deciding which objects to schedule, or when to schedule them. These decisions belong with the higher layers. It simply provides a scheduling service to the higher layers, acting upon their instructions.

An object-invocation service should also be provided to the Invocation Layer for the forwarding of invocation messages. For example :

```
Invoke(ThisObject, InvocationMessage, Options)
```

This is simply a translation service between *invocation address space* names and *location address space* names.

Management Services

The external-management services offered to the Invocation Layer concern the operational status of the scheduled objects. They should provide information on whether a particular object is still available, or whether it has become unavailable, for example, due to the failure of its associated host.

The internal-management services of the Location Layer handle the collection of host status information. Most scheduling policies require certain status information relating to each location (host) in the *location address space.* This status defines a location's 'suitability' to act as an object repository. It usually relates to performance, but could include other factors such as reliability.

Controlling object migration is also an internal-management activity of the Location Layer. Objects may migrate between locations in order to compensate

for changes in location status. Migration is a management function rather than an operational function because it is entirely hidden from the next layer up. It can be looked upon as a form of exception handling; the exception being a change in location status significant enough to cause the re-evaluation of earlier scheduling decisions.

Examples

The problem of scheduling distributed resources has been the focus of considerable research effort in recent years. There are almost as many policies as there are systems. Chapter 5 provides an overview of object scheduling, describing some of the more widely accepted techniques that belong in the Location Layer. In the example systems of chapter 1, object instances are placed at the location from which the instantiation request was generated. In most cases this corresponds to the user's workstation.

The Amoeba system, in addition to user workstations, incorporates pool processors that have no permanent owner. They are assigned dynamically to particular users when requested. The processor pool is managed by a process server that handles all requests for pool processor allocations. However, there is no defined scheduling policy concerning the allocation of objects between a user's workstation and pool processor(s). Tanenbaum and Renesse simply state that :

> "The research has not yielded any definitive answers, although it seems intuitively clear that highly interactive [objects] such as screen editors should be local to the workstation, and batch-like [services] such as big compilations should be run [on the pool processor(s)]." [Tanen85]

2.7.4 The Invocation Layer

The Invocation Layer services schedule invocation messages amongst object instances in the location-independent *invocation address space.* They also provide services for creating and deleting instances.

ORM's Invocation Layer is analogous to the OSI transport layer. The transport layer provides facilities to open a connection between any two arbitrary locations, pass data across the connection and then close the connection. This is a basic connection service, independent of the underlying network technology. It is the responsibility of the higher OSI layers to build higher level abstractions such as standard data representations and file transfer protocols. ORM's Invocation Layer services provide facilities to 'open' a service (i.e., create an object instance), pass requests to the service (using invocation messages) and then 'close' the service (i.e., remove the instance). This is a basic object support facility, independent of the underlying distributed environment. It is the responsibility of the higher ORM layers to build higher level abstractions such as virtual properties.

Naming and Addressing

Object names within the *invocation address space* are location independent. This is the first layer within ORM in which location plays no part. Multiple instances of the same object are still recognised, but they now have names such as `Object1` and `Object2`, rather than `Object@Location1` and `Object@Location2` as in the *location address space.*

The Invocation Layer creates a **Virtual Address Space** for the Virtual Layer, where only services exist. A service consists of the invocation routines provided by an object or group of (related) objects. Normally, a service will correspond to a single object. However, one-to-many and many-to-one mappings are also possible.

Service names are mapped by the Invocation Layer onto *invocation address space* names.

Operational Services

The Invocation Layer encapsulates service-provision facilities to create and remove services. For example :

```
CreateService(ThisService, Options)
RemoveService(ThisService, Options)
```

The name `ThisService` belongs to the *virtual address space*. The Invocation Layer is responsible for mapping it onto the corresponding *invocation address space* name(s). The object creation facilities of the Location Layer are then called upon to schedule the appropriate object instances.

The service-invocation facilities provided by the Invocation Layer assign invocation messages to object instances. For example :

```
Invoke(ThisService, InvocationMessage, Options)
```

If a service name maps to more than one object in the *invocation address space*, then the Invocation Layer services are responsible for selecting the instance to be invoked.

Management Services

The external-management facilities offered to the Virtual Layer should provide information on the operational status of scheduled services. For example, using these facilities the Virtual Layer might establish if a particular service is available.

As with object scheduling, invocation scheduling policies are usually based upon analysing status information. Collecting this information is an internal-management function of the Invocation Layer. The status information describes the objects in the *invocation address space*, defining their ability to respond to invocations, which usually relates to performance characteristics. Invocation schedulers assess the status information to determine which instance should receive the invocation. The invocation scheduling policy defines the criteria against which this decision is made.

Another internal-management activity associated with the Invocation Layer is the maintenance of an 'acceptable' level of service. Should the invocation rate exceed service capacity, then, where appropriate, additional object instances can be created to handle the 'overspill'. Control of the level of service by the Invocation Layer is analogous to the control of object migration by the Location Layer, i.e., they can both be viewed as exception handling. The exception in this case is an excessive workload for the current level of service provision.

Examples

In Cronus, each service type is supported by a corresponding service manager, several instances of which may be distributed throughout the system [Schan87]. Each service manager is responsible for its own group of service instances. Whenever an invocation message is generated the system kernel determines the particular service required, passing the message to any one of the corresponding service managers. The nominated service manager may then decide to pass the message to one of its own service instances or, based on status information exchanged regularly between managers, it may forward the message to a peer manager.

In Amoeba, each location runs a resource manager that handles all invocation requests directed to it by the Amoeba kernel [Tanen85]. If the resource manager has two or more suitable objects available, then it selects one at random.

2.7.5 The Virtual Layer

The Virtual Layer creates a service environment in which services possess virtual properties. Virtual properties customise a service to match the needs of its clients, for example, by making the service fault-tolerant or by presenting a security mechanism the client is familiar with. The Virtual Layer encapsulates the techniques used to create these properties.

Naming and Addressing

The *virtual address space* contains only services, i.e, not objects. The User Layer is presented with the **User Address Space** in which only virtual services exist, i.e., services that possess virtual properties. Multiple virtual services may be multiplexed onto one 'real' service, and similarly, one virtual service may be split between several 'real' services. The Virtual Layer is responsible for establishing and maintaining these mappings.

Operational Services

The User Layer should be presented with facilities for creating and removing virtual services. For example :

```
CreateService(VirtualServiceName, VirtualPropertyList, Options)
RemoveService(VirtualServiceName, Options)
```

The Virtual Layer services apply 'rules' for creating and removing these properties, mapping them onto the appropriate service creation and removal facilities of the Invocation Layer.

The service-invocation facilities in the Virtual Layer, should map invocations upon virtual services onto the equivalent 'real' services. For example :

```
Invoke(VirtualServiceName, InvocationMessage, Options)
```

For example, a fault-tolerant service can be provided by creating multiple service instances. Invocations upon the virtual service are mapped by the Virtual Layer onto multiple invocations on the 'real', non fault-tolerant instances.

Management Services

The external-management facilities offered to the User Layer should provide information on the availability and applicability of virtual properties. For example :

```
CanCreate?(VirtualServiceName, Resilient)
```

might indicate whether the specified service can be created with the resilience virtual property.

The internal-management services of the Virtual Layer should allow for the addition, removal and modification of the virtual properties made available to the User Layer. This may include maintaining a 'rule-base' for virtual properties, indicating how they are created.

The following sections describe some of the virtual property mechanisms provided by the example distributed systems of chapter 1.

Resilience

To provide fault-tolerance, Amoeba uses a boot server that periodically polls registered services to determine if they are still 'alive' [Tanen85]. If a service fails to respond properly within a specified time, the boot service declares it unavailable and initiates creation of a new service on one of the pool processors. The principle underlying the boot server is that processor crashes are infrequent, and that most

users are unwilling to pay a heavy penalty in performance in order to mask all processor crashes.

The Clouds fault-tolerance mechanism utilises primary and backup copies of each object. The backup, held at a different location to the primary, has the same capabilities as the primary but is dormant most of the time. Invocations upon the primary are also passed to the backup to maintain consistency. Periodically the backup probes the primary, expecting an '*I am OK*' message in reply. If the status report is not received before a specified timeout, then the backup takes over as the primary, creating a new backup to monitor itself.

Persistence

In the Eden system, when an eject is created only its active form exists [Almes85]. It can therefore execute and engage in invocations, but has no state on permanent store. If it were to deactivate or crash, its current status is lost forever. Eden therefore provides a checkpoint primitive allowing active ejects to permanently record their current state. An eject's stored state, known as its passive form, makes the eject persistent. Invocations subsequent to deactivation or failure of a persistent eject, cause the Eden kernel to reactivate the eject, i.e, to construct a new active form from the passive form. The reconstructed eject then receives the invocation.

Security and Protection

The Virtual Layer can manipulate the protection mechanism(s) used by services, to match those understood by clients, i.e., it can present the User Layer with the access controls expected by applications. The two principal mechanisms of capabilities and access control lists have already been introduced. The Virtual

Layer may perform conversions between the two, enabling for example, the same service to be invoked by both capability based and ACL based applications.

The Virtual Layer can also provide enhanced security by incremental addition [Karge88]. Here, the underlying security system is built upon to provide the User Layer with a more stringent level of security, for example, providing capability based access where the underlying system has no access controls. This approach to security assumes that the Virtual Layer itself is secure, and cannot be bypassed.

Proxies

The example virtual properties and associated mechanisms described so far were developed to solve specific problems in specific environments. The SOS system provides a more general mechanism through the use of proxies [Shapi86].

A proxy is a representative for one or more distributed objects that collectively provide a single service. The object(s) represented by a proxy is (are) called its principal(s). The proxy and its principal(s) form a single, distributed object known as a group. The proxy is the only interface to the group and, to the outside world, is indistinguishable from the group. The proxy, which is always local to its client, provides a single entry point to a potentially distributed service; all invocations upon the service being made through the proxy. Invokers are unaware of the proxy's existence, believing they are invoking the service directly.

Proxies belong in the Virtual Layer because they mediate access to groups of services, manipulating the facilities they offer to provide some 'bigger' or 'better' service; a description that fits exactly the role defined for Virtual Layer services. Some of the more interesting properties of proxies are [Shapi86] :

1. *Encapsulation.* The service is a black box, accessible only through the proxy. Its structure is not exposed.

2. *Access protocol.* The proxy enforces per-client ordering constraints on calls (e.g. enforce a request-acknowledgement-access-release ordering on the use of a resource).

3. *Capability.* The proxy can implement access controls, test the validity of arguments, or the right to perform certain operations; it is totally programmable.

4. *Protocol encapsulation.* The protocol between the client and the service is totally encapsulated within the distributed object formed by the proxy and its principal(s).

The philosophy promoted by proxies, that of a general mechanism for coordinating and enhancing services, will be returned to later in the thesis.

2.7.6 The User Layer

As the highest layer in the Object Reference Model, the User Layer provides services to users and applications, rather than to another layer. It provides users with access to a 'world' of virtual objects, creating an environment within which users and applications can operate.

Naming and Addressing

The *user address space* seen by users is created, as described above, by the Virtual Layer. Only virtual services exist within this address space. From the users' point of view, there are no hosts, no distribution, no network and, upon application of the appropriate virtual properties, no failures, data inconsistencies or security problems.

The User Layer's role in naming and addressing is therefore not in providing the *user address space*, but in managing it. The nature of the systems under discussion mean they will have many (hundreds of) users, and consequently many (potentially millions of) objects. In general, an individual user will only be interested in a small proportion of these objects. The User Layer must therefore provide facilities for organising and accessing the many objects available.

Operational and Management Services

The User Layer creates the object environment observed by users and their applications, providing access to all available services. The nature of the User Layer makes it difficult to predict the operational and management services required. However, the following list provides a few general examples :

- Service directory listing
- Service creation and deletion
- Service invocation
- User environment customization such as name aliases
- User environment defaults for virtual properties

User requests for these services, coupled with the appropriate management information, must be converted, by the User Layer, into requests upon the service provision and management facilities of the Virtual Layer.

Example

The Amoeba architecture is based upon a series of layers, the highest two of which provide the type of services modelled by ORM's User Layer [Tane81b].

Amoeba's system call layer provides user programs with a traditional operating system interface. It supplies a number of routines that users invoke to provide the services offered by most timesharing systems. These routines operate by sending requests and getting replies from appropriate services in the network. In most Amoeba implementations this layer is a library package designed to emulate some particular set of operating system calls. Amoeba users' programs run in Amoeba's user layer, built on top of the system call layer. Most programs use the system call layer to provide a simple and familiar environment in which to run.

2.8 Summary

The Object Reference Model (ORM) provides a conceptual framework for describing object oriented distributed systems. As such it can be used to assist both in the development of new systems and in the analysis of current architectures. The latter role has already been demonstrated by the examples provided throughout this chapter. The former role is employed throughout the rest of this thesis.

ORM was developed by applying the OSI layering principles to a list of key design problems identified after extensive analysis of many current systems. As a result, six layers were defined (Figure 2–3) :

The **Migration Layer** provides a network independent communication facility, creating the communication services required by the higher layers. Typical services include object migration, and invocation protocols based on remote procedure calls. The Migration Layer presents the Construction Layer with the *construction address space*, where objects reside at communication independent addresses.

The **Construction Layer** provides a transformation facility for manipulating object representations. It encapsulates the inherent heterogeneity of the underlying environment, masking it from the higher layers. The Construction Layer presents

the Location Layer with the *location address space* in which objects are referred to by generic names, with no awareness of multiple representations.

The **Location Layer** controls the assignment of object instances to *location address space* locations. It encapsulates the location dependencies inherent in a distributed environment, masking them from the higher layers. The Location Layer presents the Invocation Layer with the *invocation address space* in which only service names exist, with no concept of location.

The **Invocation Layer** builds upon the services of the lower layers to provide a basic object environment in which objects can be created, invoked and deleted. Its principal function is controlling the assignment of invocation messages to object instances. The Invocation Layer presents the Virtual Layer with the *virtual address space* in which only services exist. The mapping between services and object instances can be one-to-one, one-to-many or many-to-one.

The **Virtual Layer** embodies the techniques used to create virtual properties. A virtual property is a characteristic, possessed by objects, that somehow 'improves' the service offered, but in a service independent manner. Typical examples are fault tolerance and persistence. The Virtual Layer presents the User Layer with the *user address space* in which only virtual services exist, i.e., services possessing virtual properties.

The **User Layer** is the uppermost ORM layer. It creates the environment in which users and their applications operate, providing user access to all available services.

Each layer builds upon and adds to the services of the layer below. The uppermost layer provides users and applications with a location independent, homogeneous environment in which services possess 'desirable' (i.e., virtual) properties.

The remainder of this thesis examines resource provision in object oriented distributed systems, based upon the framework defined by the Object Reference Model.

Chapter 3

The Target Environment

Having described the general Object Reference Model, this chapter now defines the specific target environment addressed by the remainder of the thesis. Three categories of distributed hardware are identified — workstations, pool processors and specialist processors — although ultimately these are all seen simply as hosts for objects. The minimum required network capabilities are defined. An overview of the three leading local area network technologies is included; namely Ethernet, Token Bus and Token Ring. The limited role of wide area networks is also discussed. A system-wide Invocation service is assumed to be available, based upon remote procedure call. Encapsulation is stated as being the minimum object oriented feature required of object implementation languages. Finally, assumptions concerning the user population and envisaged applications are stated, concluding with a discussion on how these affect the system design.

3.1 Introduction

The purpose of this chapter is to define the distributed environment addressed by the remainder of the thesis. All assumptions concerning the general nature of the environment are stated. The flow of discussion is based upon the system components shown in Figure 2–1 (page 35). Familiarity with the Object Reference Model is assumed throughout.

3.2 Distributed Processors

From a resource provision point-of-view, all distributed processors are simply locations at which objects can be hosted. However, the computing power of the target environment can be divided broadly into three categories; workstations, pool processors and specialist processors, each of which is discussed below.

3.2.1 Workstations

The distributed processors in the target environment are assumed to comprise mainly of general purpose 'personal' workstations. Workstations are only considered personal in the sense that only one user at a time may access the console. However, the computing resources offered by workstations are not considered personal, they simply form part of a resource 'pool' managed by the system. Hence, it is assumed the system is free to distribute objects at its own discretion, unrestricted by user 'ownership' of machine resources. No specific upper limit is placed on the number of workstations, but it is assumed to be several hundreds. This limit depends primarily on the ability of the underlying network to support the

communications traffic generated (see section 3.3 below). The workstations are not assumed to be homogeneous.

Workstations generally exhibit the following characteristics :

- ▷ 'Powerful' processing capability
- ▷ 'Large' memory
- ▷ Simultaneous support for multiple object instances (multi-tasking)
- ▷ High resolution displays
- ▷ Support for iconic, window and mouse based interface

The first three items relate to system capacity, i.e., the quality of service the system is capable of providing. The last two items relate to the user interface. An iconic interface providing user access to the object environment is an example of an ORM User Layer service.

3.2.2 Specialist Processors

Specialist processors provide specialist resources, usually implemented in hardware, over and above those offered by workstations. Depending upon the nature of its specialisation, a specialist processor may simply be better than workstations at providing certain types of service. Alternatively, it may provide one particular service to the exclusion of all others; usually at a capacity considerably greater than achievable by general purpose workstations.

Examples of specialist processors include peripherals such as file stores and laser printers, parallel machines such as array processors and pipelined processors, and processors with specialist architectures such as dataflow machines and systolic

arrays. All specialist processors must present an object oriented interface to the distributed environment. Section 2.2.1 discussed this requirement, concluding that it can always be met, either by supporting an object instance directly or by using a front-end. Using front-ends may incur additional resource costs, but this is assumed to be offset by the increased utility provided by access to the specialist services.

Specialist processors therefore simply represent further (specialised) locations at which objects may reside.

3.2.3 Pool Processors

Within the target environment, pool processors are any non-specialist processors that do not offer support for a user interface. As stated earlier, the processing resources offered by workstations are not retained exclusively for use by the workstation's owner. As such, workstations can be viewed simply as pool processors that have a screen, keyboard and mouse attached. Using the terminology introduced above, the screen, keyboard and mouse are specialist processors front-ended by a window manager running on the associated workstation processor.

Pool processors therefore simply provide additional locations at which object instances can be placed. As with workstations, there is no assumption that pool processors are homogeneous.

3.2.4 Object Hosts

Workstations, pool processors and (some) specialist processors represent locations at which objects can be run, i.e., they host objects. An **Object Host** is any object capable of interpreting object representations (at some level of abstraction) and performing the actions they describe. Note that hosts are themselves defined to be

objects. Therefore workstations, for example, appear as object instances supporting invocation routines such as `Host(ThisObject)` and `UnHost(ThisObject)`.

Usually, a host corresponds to an individual workstation, pool processor or specialist processor. However, many-to-one and one-to-many relationships are possible. For example, multiple hosts, potentially interpreting different representation abstractions (virtual processors), may be implemented on a single workstation. Alternatively a single host may be implemented across multiple workstations. This latter configuration is in fact the goal of the Object Environment; to present users with a single, universal host implemented across all workstations, pool processors and specialist processors in the distributed environment.

The recursive definition of hosts — hosts implemented as objects supported by hosts — terminates at the hardware level when, the host implementation becomes self-supporting. This can be considered as the 'base case' of the recursion, i.e., the 'real' host upon which other, 'virtual', host abstractions are built.

3.3 Network

3.3.1 Technology

It is assumed that a fast, reliable transmission facility is provided by the underlying network. The propagation delay between any two hosts is also assumed to be uniform across all hosts, i.e., no host is significantly 'further away' than any other. Local area networks meet these requirements significantly better than wide area networks. Even in LANs the propagation delays depend upon the sender's and recipient's relative positions. However, this variation is sufficiently small that the above assumption is still deemed to be satisfied.

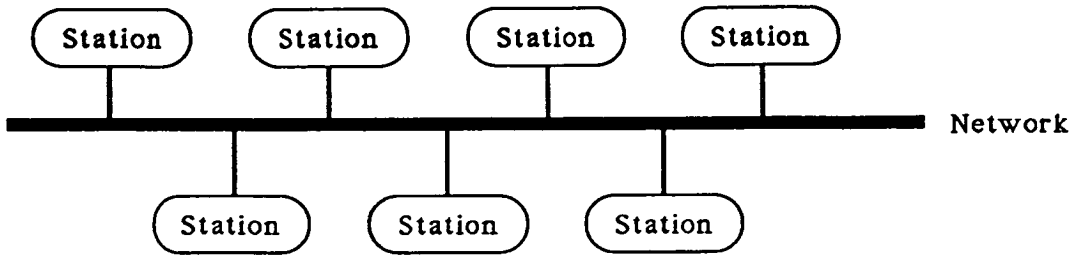

Figure 3–1: Ethernet Simple Bus Topology

The leading LAN technologies have been standardised by the Institute of Electrical and Electronic Engineers' (IEEE) 802 committee. They have defined a family of standards incorporating the Contention Bus (IEEE 802.3), the Token Bus (IEEE 802.4) and the Token Ring (IEEE 802.5). All three provide a suitable basis for a distributed system. The following sections provide an overview of these networking technologies. Where there is no danger of ambiguity, the term **Station** will be used in the following discussion to collectively denote workstations, pool processors and specialist processors.

Contention Bus

The IEEE 802.3 contention bus standard is based upon Ethernet. Ethernet is a local area network developed initially at Xerox PARC as part of an extensive research programme on personal workstations and distributed systems. It offers a data rate of 10 million bits per second to up to 1024 stations, with a maximum station separation of 2.5 kilometers [DIX 80]. The Ethernet bus topology is shown in Figure 3–1.

When a station wishes to communicate, it listens to the network to see if any other communication is currently in progress. If not, it attempts to transmit. In most cases the transmitted message will be delivered unhindered to the recipient station. There is however, a possibility of message collision caused by two stations

simultaneously observing a quiet network, and attempting to transmit at the same moment. Therefore, during transmission, the transmitting station continually monitors for collisions. Should a collision be detected then transmission ceases immediately. Retransmission is attempted only after a random timeout in order to avoid repeated collisions. This technique is known as carrier sensing multiple access with collision detection; generally abbreviated to CSMA/CD.

Token Bus

The IEEE 802.4 LAN standard defines a token-passing bus protocol. The network topology is the same as for Ethernet (Figure 3–1), but, the stations are ordered to form a logical ring, each station having a predecessor and a successor. A token is passed between the stations, following the sequence of the logical ring. The current token holder acts as a master station for the time it holds the token, giving it exclusive access to the bus, and enabling it to transmit. When the station has completed transmission, or when its time has expired, the token is passed on to the next station in the sequence. In this way, access to the bus is provided on a round-robin basis, with each station guaranteed a chance to transmit.

The length of the ring, coupled with the timeout on token ownership, provides an upper limit to the time a station must wait before gaining access to the network. The token bus protocol therefore provides deterministic access times, making it suitable for real-time applications. This is at the expense, however, of complex token management that has to cope with : additions to the ring, with new stations periodically being given the opportunity to enter the logical ring; deletions from the ring, for stations wishing to 'opt-out'; removal of duplicate tokens, which otherwise lead to bus collisions; detection of failed successors, to ensure the ring remains unbroken; and ring initialisation, to start a token circulating either at network start-up, or after an error causing it to be lost.

Token Ring

In ring topology networks the stations are physically connected in a ring (Figure 3–2), with each station forwarding data from its input stream to its output stream. Data therefore travels around the ring in a specific direction. In a token ring network such as that specified in the IEEE 802.5 LAN standard, a single token circulates continuously around the ring, marked with one of the two possible states 'free' or 'busy'. A station wishing to communicate must wait until the token reaches it in the free state. The token is marked as busy, the destination and source addresses are added, and the data to be transmitted are attached. Arbitrary length data packets are permitted, up to some pre-defined (installation dependent) maximum. The token continues around the ring until reaching the destination station, which then copies the contents of the message and flags the token as 'received'. The token then continues around the ring, eventually returning to the sender. After checking the token's flags to ensure it was received successfully, the sender removes the message and inserts a new 'free' token into the ring. Thus, the next station in the ring has an opportunity to seize the token and transmit a message.

The effect upon the target environment of using these network technologies will be assessed later.

Wide Area Networks

Wide area network links are generally slower and more error prone than local network communication. Such links are sometimes used to merge multiple LANs, creating the impression of one 'large' local network. For example, the Universe Network [Lesli84] connects ring networks at various sites in the U.K. by a 1 Mbit/s satellite broadcast channel.

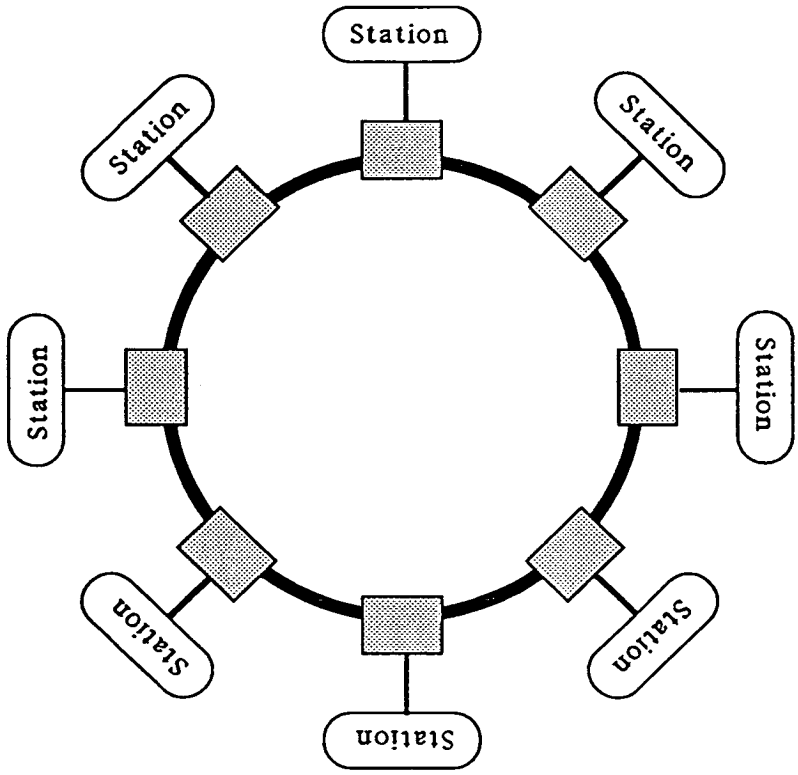

Figure 3–2: Ring Topology

The target environment does not attempt to invisibly merge geographically dispersed local area networks. Wide area links are assumed only to be used in providing limited access to resources not available on some local networks.

3.3.2 Protocols

Broadcasts

In a bus topology network, all stations monitor all transmissions, looking for a destination address that matches their own. In a ring topology network, every messages passes through every node, again with each node looking for a match between destination address and its own address. Having reached its destination, each message still continues around the ring until it returns to the sender, which removes it from the network. Hence, the three LAN technologies outlined above — in fact, most LAN technologies — provide broadcast facilities whereby a single message can be communicated to all stations with little or no additional overhead compared to one-to-one communication. This facility, which can be useful for propagating status information and locating services with 'WhereIs?' messages, is therefore assumed to exist within the target environment.

A variant of broadcasting, known as multicast, provides a general one-to-many communication facility rather than simply one-to-all. Although useful, this is not universally available and is therefore not relied upon.

Communication Services

The IEEE 802 networking standards define protocols up to the level of the OSI Network Layer (Figure 3–3). The transport, presentation and application functions are not included as part of the basic services offered by these networks. Effectively, these protocols provide a transmission service, but they do not provide

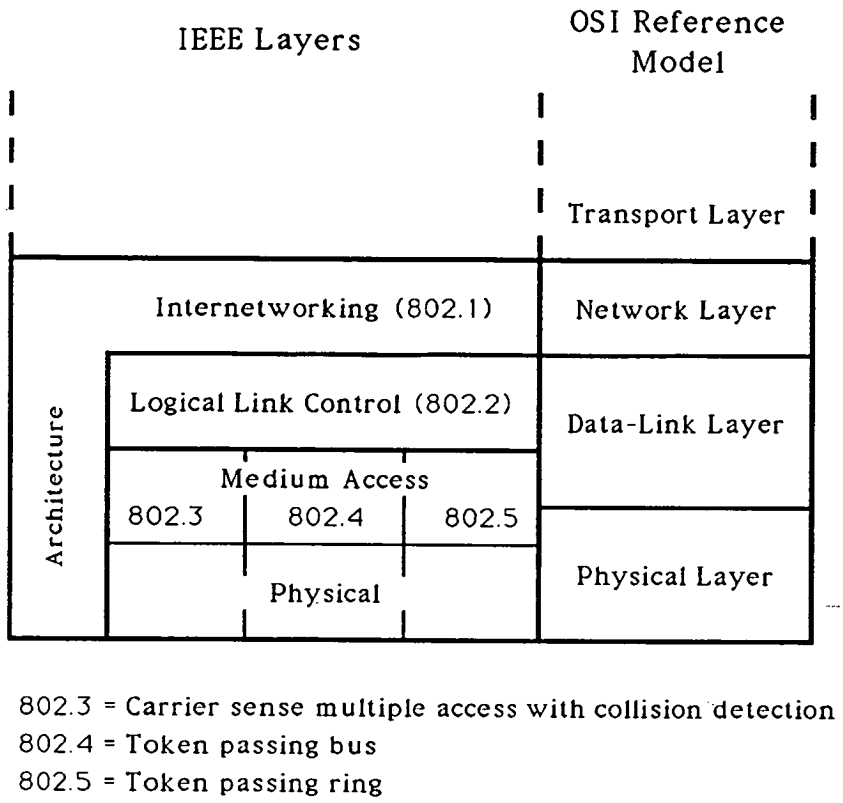

Figure 3–3: IEEE LAN standards and the OSI model

the high level *communication* services required by the target environment. Higher level protocols are therefore also assumed to be provided by the underlying communication system, implementing services such as standard data representations and remote procedure calls.

Remote Procedure Calls

Considerable research effort has been directed towards the communication services required for distributed environments, since these services are fundamental to building any form of advanced distributed system. In particular, experience with the use of remote procedure calls has led to the development of stubs and stub generators, making RPC easier both to implement and use.

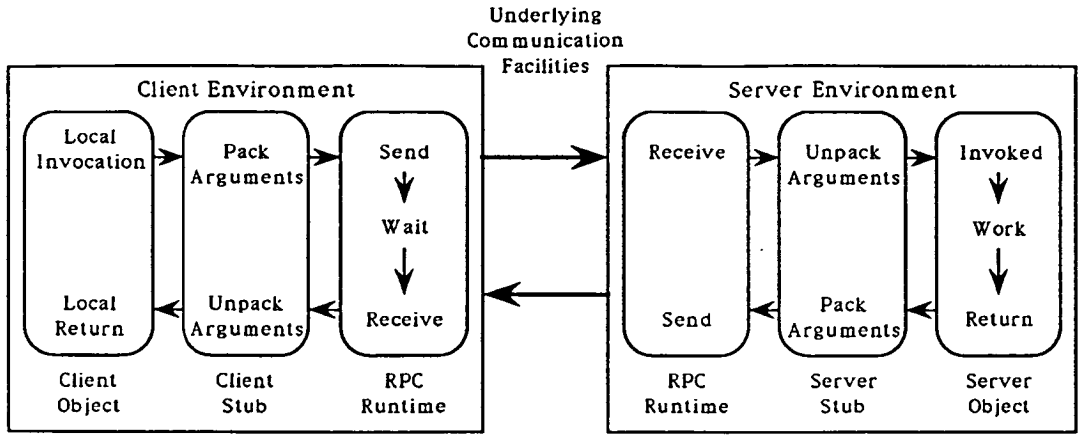

Figure 3–4: Remote Procedure Call using Stubs

Figure 3–4 demonstrates the interaction of RPC stubs for a simple call. The client makes a local invocation, which is intercepted by the client-stub. The client-stub constructs the appropriate communication 'packets', marshalling the invocation message and its parameters into a form suitable for the underlying network. The invocation message is then passed to the RPC runtime system, which forwards it to the remote destination. The remote RPC runtime system receives the invocation message, passing it to the server-stub, which unmarshalls the parameters and makes the appropriate (local) invocation upon the server object. Results are returned in the same manner.

Several systems provide automatic generation of client and server stubs, producing customised stub routines for each service. Three examples, discussed below, are the Mach System, the University of Washington's Heterogeneous Computer System (HCS), and SOS. These systems allow client and server objects to be implemented as though for a non-distributed environment, i.e., without regard for the problems of remote addressing, parameter marshalling and network communication. These distribution aspects are handled jointly by the RPC runtime system and the object construction mechanisms.

Mach's Matchmaker [Jones86] allows interfaces between cooperating objects to

be specified and maintained independent of specific languages or machine architectures. The Matchmaker code generated from interface specifications provides communication, type conversions, runtime support for type-checking, synchronization and exception handling. It currently supports the languages Common Lisp, C, Ada and Pascal. Both clients and servers may be built in any of these languages.

The HCS project provides a set of 'black box' RPC components offering 'mix 'n' match' RPC protocols [Bersh87]. The set of protocols to be used is determined dynamically. Therefore, multiple instances of the same object may co-exist, each using different stubs and potentially different RPC runtime protocols. This approach retains compatibility with existing protocols without constraining new developments in RPC research.

The SOS system provides a generic RPC proxy that simply reproduces locally the interface presented by a remote service, blindly forwarding all local invocations to the remote object [Shapi86].

In light of the widely reported success of using the remote procedure call paradigm in distributed environments, the communication services are assumed to include system-wide, uniform, RPC facilities. These services are assumed to provide the Migration Layer with the basic facilities necessary to create a uniform invocation mechanism.

3.4 The Object Environment

The general flavour of facilities offered by the Object Environment were extensively reviewed in chapter 2. This section identifies features specific to the target environment.

3.4.1 Migration Layer Facilities

The Migration Layer services are assumed to provide a system-wide, uniform invocation mechanism based upon the remote procedure call facilities provided by the underlying communication service. At the client end, invocations destined for remote hosts must be mapped onto the appropriate RPC service(s). A remote call is then made to the server host by the communication services. At the server host the incoming RPC must be mapped to the appropriate (local) invocation. Each host must therefore provide facilities for mapping invocations both to and from the remote procedure call facilities. It is assumed that each invocation message is time-stamped by the recipient host upon arrival. It is further assumed that most invocations are synchronous, i.e., the client is suspended until the server returns its results. Asynchronous invocation, where the client continues to function while the invocation is serviced, introduces additional complexities in returning replies to clients that are now performing other activities.

As well as providing an invocation service, the Migration Layer is responsible for object migration. Later chapters argue that object migration for the purpose of load balancing is not the most efficient technique, and that other mechanisms are more appropriate. The provision of highly efficient migration protocols for load balancing is therefore not considered important. However, object migration facilities are still required, for example, to implement the object invocation service outlined above, and for passing object representations between distributed

hosts. These are essentially data transfer protocols and are should therefore be straightforward to implement using the underlying network communication facilities. Hence, the target environment is assumed to have available an object migration service.

3.4.2 Object Orientation

The work described in the remainder of this thesis utilises object oriented features to address resource provision problems. The features exploited most are encapsulation, and communication by invocation messages. On some occasions, inheritance is also suggested as a useful tool for solving certain problems. However, inheritance is not relied upon heavily and is therefore not a strong requirement. This implies that the target environment can incorporate encapsulated services implemented in non-object oriented languages [Methf87]. The invocation facilities, and in particular RPC using stub generators, are assumed to provide an object oriented interface to non-object oriented languages such as C and Pascal.

For example, to implement an object, the C programmer creates a library module with the services coded in C. The object construction, invocation and RPC support facilities 'dress' the C module to present the distributed environment with an object instance providing exactly the routines contained in the module. This is the same approach as described earlier in relation to the Mach system. Obviously, languages supporting object oriented features such as class definitions and inheritance also fit into this framework.

3.5 Applications and Users

The nature of large scale computing resources makes it difficult, and perhaps even unwise, to predict the uses to which they will be put. No particular restrictions are placed on the target environment in terms of the number of users or kinds of applications.

3.5.1 User Population

The maximum size of a system's user population will, ultimately, be determined by the number of processors in the environment. The Ethernet is capable of supporting up to 1024 stations, while the ring networks, in theory, can be extended to incorporate an arbitrary number of stations. However, in practice, they are limited to the same order of magnitude as Ethernet. The processor population in any particular system therefore has an upper limit of around 1000. The maximum active user population is therefore also limited to same amount. However, users can gain access to the system only via workstations, and not all processors are workstations. This further limits the maximum number of users. Also, since one of the justifications for creating a distributed environment is to provide users access to more and varied resources than can be provided by a single workstation, the relationship between users and processors is not one-to-one.

The maximum possible user population that could be encountered is therefore less than 1000. In practice, since not all distributed systems have available, or need, the largest permissible number of stations, the maximum user population is expected to be around 200 to 300 users.

3.5.2 Application Performance

Although users justifiably expect 'good' performance from the system, absolute speed is not seen as the top priority. For example, squeezing the maximum parallelism from a particular problem is a specialised task not suited to the general facilities envisaged here. Applications requiring this sort of parallelism should be scheduled on the specialist parallel processors within the system.

The management and exploitation of parallelism for performance is not an explicit goal of the target environment. Parallelism at the system level is provided 'free', simply because of the multiple, distributed processors. Parallelism at the user level, i.e., with more than one processor working on an individual user's task, is dependent upon the distributed scheduling policy. The scheduler's main objective is simply to make efficient use of available resources. However, parallelism may result as a by-product of efficient distributed scheduling.

3.5.3 System Optimisation

The nature of possible applications for the target environment is wide-ranging. The SOS system, for example, is biased towards office automation, principally the handling of multi-media documents. The higher levels of the system can therefore be optimised to perform specific tasks well, namely the storage, retrieval and manipulation of documents. The Amoeba system, and others, are biased more towards scientific and engineering computing. With these applications there is less scope for specific optimisation as the system is used as a general purpose tool, handling a wider range of tasks.

The target environment is assumed to provide a general computing facility, more akin to Amoeba than SOS. The expected applications are general purpose rather than being limited to a few specific tasks. This assumption limits the scope for application optimisations to the ORM User Layer, and possibly the

Virtual Layer. For example, every application could be presented with its own customized environment. More realistically perhaps, several different application environments may be provided, each optimised towards certain types of service. Optimisations to the lower ORM layer protocols cannot be guaranteed to be universally beneficial.

Chapter 4

Object Construction

The problem of automatic object construction is addressed, leading to the development of a distributed construction algorithm. A data structure, known as a construction graph, is introduced to describe the relationship between different representations of an object. A construction path is a route through the construction graph defining the transformation necessary for converting the 'source' object representation, into the required 'destination' representation. A distributed search algorithm is presented for finding suitable construction paths and creating objects, operating without the need for user intervention. This algorithm provides a simple construction service for user applications, fulfilling the requirements of the ORM construction layer.

4.1 Introduction

The purpose of ORM construction layer services was defined in chapter 2 as :

> Providing a transformation facility for manipulating object representations, thereby encapsulating the inherent heterogeneity of the underlying environment, masking it from the higher layers.

This chapter examines the potential for providing an *automatic* object construction service, i.e., a construction service not dependent upon user intervention. The following section argues that automatic manipulation of object representations in a heterogeneous environment is both desirable, and necessary.

4.2 Automatic Construction

4.2.1 Natural Progression

The earliest computers were programmed in binary or by placing wire jumpers in a patchboard. This form of 'programming' suited the machines, but was very inconvenient for human users. Assembly languages and simple compilers were soon developed. These changed the method of program description to a form more suited to humans than machines, with conversion tools (the compilers) used to translate between the different representations. In current programming environments, programmers work almost exclusively with 'soft-machines', i.e., high-level, abstract programming languages far removed from the underlying hardware. The trend in language development has always been towards ever greater levels of abstraction.

Despite the advances in language design, users remain aware of machine requirements because of the need to always compile programs before they can be run. The desire for automatic object construction therefore presents itself as a natural extension to the development of programming language design. If desired, programmers should be able to work with a high level abstract machine capable of directly interpreting their abstract programs.

4.2.2 Proven Feasibility

Smalltalk-80 is a single user, non-distributed, integrated programming language and programming environment developed by the Software Concepts Group at Xerox PARC [Goldb83]. Its designers expressed the sentiment that :

> "People work with problem-domain concepts, while hardware works with different (operator/operand) concepts. Some of the conceptual burden in translating from problem-domain to computer-domain should be carried by the machine, by making the machine work in terms of concepts closer to the user's everyday world." [Cox 87]

Smalltalk-80 provides an integrated environment in which the programming language, supporting tools (text editors, debuggers), and the operating system itself form a coherent, tightly coupled whole [Goldb84]. Smalltalk-80 methods (invocation routines) are implemented in the Smalltalk-80 programming language. The underlying hardware executes the Smalltalk-80 virtual machine, interpreting Smalltalk-80 bytecode instructions. The transformation between high level language statements and their corresponding bytecodes is performed automatically.

In the Smalltalk-80 environment, when a new method is created or an existing method is modified, the user must `Accept` it using a drop-down menu controlled by the system mouse. From a user point of view this is simply stating "I've

finished coding this method". At this point the system compiles the method and, providing there are no syntax errors, adds the compiled method to the associated class's method dictionary[1]. The compilation itself remains invisible to the user. In order to invoke a method, the user highlights the method name using the mouse, and selects the option `Do it` from a drop down menu. Arbitrary segments of Smalltalk-80 source code can also be executed in this manner.

The Smalltalk-80 programming interface demonstrates that program compilation can be successfully and usefully hidden from users, who remain unaware of any representation other than Smalltalk-80 expressions.

4.2.3 Configuration Flexibility

Ideally, the object scheduler should be free to schedule object instances on any host. To satisfy this in a heterogeneous environment requires many different object representations; potentially as many as there are different hosts. If control over the generation of object representations is left with users, the scheduler's options are restricted to the representations supplied. Hence, under these circumstances, user actions could become significant in influencing scheduler performance. Restrictions of this nature are considered undesirable. Automatic construction of executable object representations is therefore regarded as an important prerequisite to effective object scheduling.

---

[1]Even within a single class, methods are created, edited and compiled individually.

4.3 Requirements

The general requirements for an object construction service were outlined in chapter 2 when describing the ORM Construction Layer. The principal service is the creation of object representations suitable for execution at specified locations within the *construction address space*. These locations correspond precisely to the object hosts of the target environment. The construction service is also responsible, where necessary, for transforming invocation parameters into a representation understood by the invoked object. These services require controlling the application of transformation tools such as compilers, linkers, code generators and program translators. The intention is to provide an integrated environment similar in spirit to that of Smalltalk, but for all languages within the system. The user environment should appear to interpret directly high level programs, removing the need for explicit, user initiated compilation.

The external-management services identified in chapter 2 should also be provided. These supply management information on possible transformations, ideally including some measure of associated cost. The purpose of this management information is to assist in object scheduling decisions. This is the only role the construction facility has in connection with scheduling. The scheduling decisions themselves are made by ORM Location Layer services.

This thesis does not propose a new approach to language design or compiler technology. Rather, it presents a distributed algorithm for automatically controlling the application of existing object construction tools.

4.4 Limitations and Assumptions

4.4.1 Construction Limitations

The proposed mechanism is intended only for constructing user application programs, i.e., programs existing within the user environment. It it is not intended to manage code implementing the system itself, which is considered to be a separate problem handled by the system implementors using traditional (manual) techniques. The system code implementing the construction service, and in particular the transformation tools themselves, are therefore assumed to be pre-constructed. This removes the possibility of recursive invocations to 'construct the constructor', thereby allowing invocations upon transformation tools to be scheduled using the standard invocation scheduling mechanisms.

4.4.2 Error Free Syntax

Services in the ORM User Layer are assumed to handle the user oriented aspects of programming by providing a development environment for each language, incorporating, for example, tools such as syntax directed editors and syntax checkers. As an example, one of the environments offered by the application layer might be a Smalltalk-80 emulation running on top of the target environment. Separating the user interface from the mechanics of object transformation, simplifies the transformation tools, allowing them to perform faster.

The object construction service therefore deals exclusively with object transformations. Issues such as creating source code and ensuring correct syntax are assumed to be handled by services at the User Layer level. The transformation tools therefore assume all object representations are syntactically 'correct', having

been previously vetted. For example, an object purporting to be a C++ implementation of some service is guaranteed to conform to C++ syntax. It is further assumed that all necessary parameters, such as the objects to be included in a link operation, are readily available. These can be obtained either from user supplied information (interrogation being performed by the User Layer environment), or by automatic deduction and dynamic searching (again, performed by User Layer services rather than the construction service).

4.5 Object Transformations

In current systems, an object implementation, or indeed an implementation in any programming paradigm, is transformed several times before a representation is derived suitable for execution on the system hardware. As an example, the simulation program introduced later in the thesis was implemented in Simula. The first step towards generating an executable representation of this program is the application of the Simula compiler. The compiler generates an assembly-language version of the program, which is then further transformed by an assembler. The final stage is performed using a linker to merge the output from the assembler with the necessary library and system code. The relationship between the different program representations is shown in Figure 4–1.

The simulation was also run under a different operating system using the same source code, but with a different compiler and linker. These new representations can also be added to the relationship diagram, as shown in Figure 4–2. In this case the diagram indicates that no intermediate assembly language representation is produced. Many other executable representations could be produced from the same source code, each of which can be added to the relationship diagram in a similar manner.

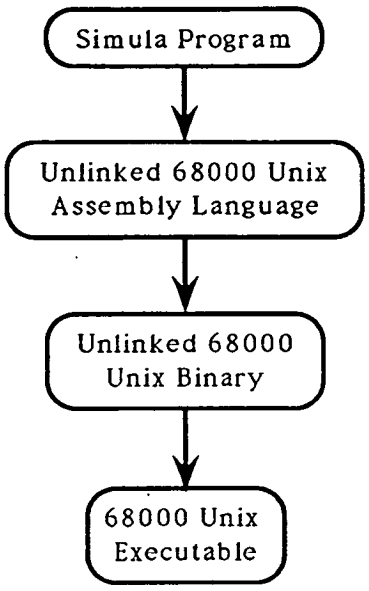

Figure 4–1: A Simple Representation Relationship Diagram

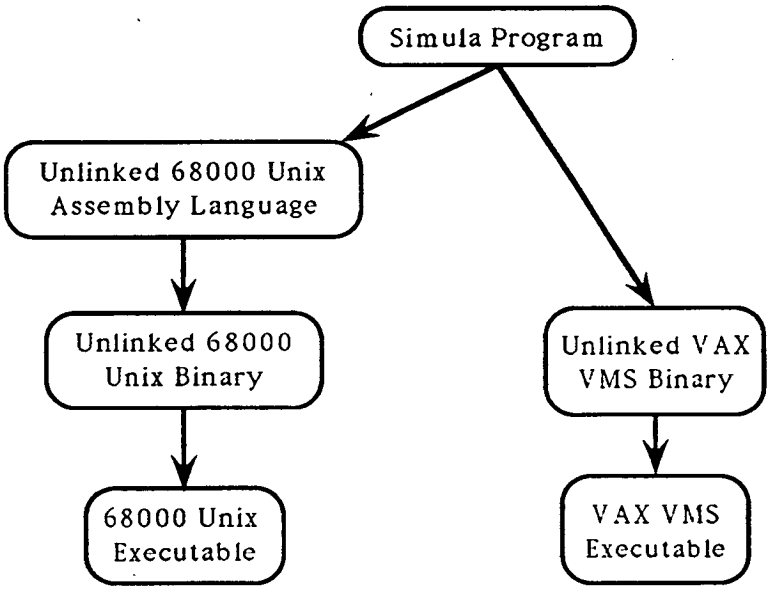

Figure 4–2: An Extended Representation Relationship Diagram

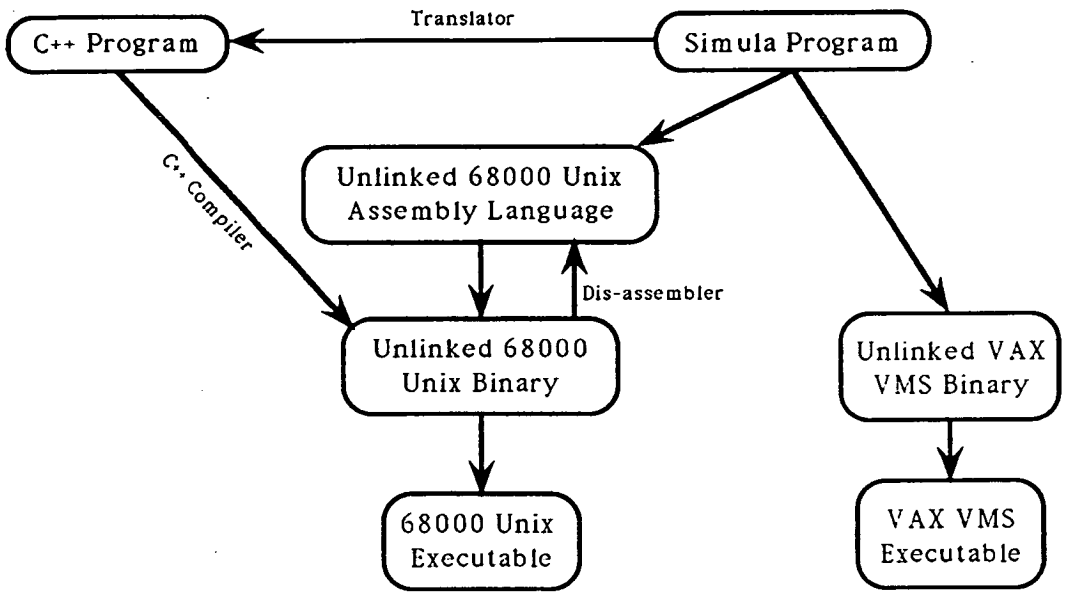

Figure 4–3: A Hierarchy of Program Representations

The transformations described above move from a higher to a lower level of abstraction. This is not always the case. For example, dis-assemblers perform the reverse function of assemblers, producing as their output a representation at a higher level of abstraction than their input. Another example is provided by program translators that transform one high level language implementation into another, e.g., C to Pascal, or Simula to C++. Figure 4–3 extends the relationship diagram to include a Simula to C++ translator and a dis-assembler.

4.6 Construction Graphs

Relationships of the type described by Figure 4–3 are based on representations rather than any object specific characteristics. Consequently, they are applicable to all objects. This graphical representation of object construction can be generalised to include all possible representations within a system, identifying the relationships between them. Graphs of this nature, known as **Construction Graphs**,

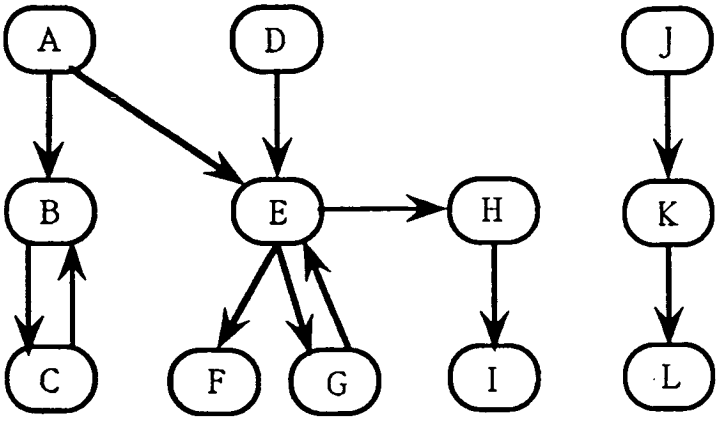

Figure 4–4: A General Construction Graph

can be thought of as state transition diagrams, where the states correspond to object representations, and state changes are triggered by the application of a transformation tool. Figure 4–4 shows a general example of a construction graph with representations denoted A, B, C and so forth, rather than specific representation names such as Simula, C++ and assembler. Note that the graph does not have to be totally connected. As in the example, no transformation tools are available to connect the sub-graph J-K-L, to the rest of the graph structure.

Each node in the construction graph corresponds to a particular representation or **Type**, while the edges represent the possible transformations. The nodes are termed **Transformation Nodes** since each one encapsulates the system-wide knowledge concerning transformations applicable to their corresponding types. Thus, the Simula transformation node '*knows all things that can be done with Simula objects*'. Similarly the 'A' transformation node '*knows all things that can be done with object representations of type A*', and so on.

Every object transformation involves at least two representations, the **Source Type** and **Destination Type**, corresponding respectively to the pre- and post-transformation representations. When constructing an executable representation,

the destination-type will correspond to the **Host Type**, i.e., the representation expected by the specified host.

A destination-type can only be derived from a specified source-type if there exists a **Construction Path** between their respective transformation nodes, i.e., a series of one or more transformations converting the source-type into the destination-type. The problem of object construction can therefore be reduced to that of finding construction paths through the construction graph. An object is constructed by **Traversing** its construction path, i.e., moving sequentially along each arc in the construction path, from the source node to the destination node, invoking the appropriate transformation tool at each step.

4.7 Searching For Construction Paths

Construction graphs describe the relationships between object representations, identifying the transformation tools used to convert between them. For any given source and destination-type a construction path must be found linking their respective transformation nodes. This requires searching the construction graph.

The envisaged client of the construction service is the object scheduler, which knows only about objects and object locations in the *location address space.* More specifically, the scheduler does not know about source and destination types. The object scheduler therefore expects a construction service interface such as :

```
MakeInstance(Object, @Location, Options)
```

Hence, the construction service only knows the destination-type, which it derives from the specified location. The source-type is, initially, unknown. The starting point for a construction path search is therefore the transformation node of the destination-type. The search moves 'backwards' from the destination-type, looking

for one or more, source type representations of the specified object. Note that this is contrary to the normal, user driven method of object construction in which a source representation is known, provided by the user.

A construction path is found by searching the construction graph for a source representation of the object. Using a breadth-first search, starting at the destination node, the 'closest' source will be encountered first, i.e., the source representation requiring the least number of transformation steps to create a destination-type representation. For example, if an intermediate representation of the object already exists, then the search will find this and terminate, never reaching the original high level language representation. If a destination-type representation already exists, then the search terminates immediately. In this case the construction path is reduced to a single node, with no transformations required.

The `Options`-parameter included in the `MakeInstance` service may serve several purposes. For example : it may specify a preferred construction path if more than one source is available; it may specify a particular path, overriding the construction service's choice; when more than one transformation tool is available for a particular transformation, e.g., two C++ compilers, it may specify which one to use; finally, it may also specify 'flags' for the transformation tools, for example, setting a `Debug` flag.

4.8 Representation Cacheing

As hinted by the previous section, and suggested by common sense, objects need not be reconstructed from scratch every time they are required. When an object transformation is performed, the resultant representation can be **Cached**, i.e., stored for future use. In subsequent requests to construct the same object, the cost of transformation is replaced by the (significantly smaller) cost of locating and retrieving the cached representation.

The improved construction performance gained through cacheing must be traded against the cost of managing and storing the cached representations. Many different cacheing policies could be envisaged, ranging from none at all, to cacheing everything. A simplification of the cache-all scheme would be to cache only host-type representations, i.e., the end result of a series of transformations, discarding any intermediate representations produced on the way. Another policy could use frequency of access, for example, only cacheing an object if it has already been constructed n times within some time-period. Other policies could be based on privileges, such as only cacheing an object if it has 'cacheing privilege'.

These example policies are by no means exhaustive. As with most forms of cacheing there is no 'best' policy, only compromise, dependent upon many environment specific factors. Policy recommendations for object cacheing are therefore beyond the scope of this discussion. Object cacheing is secondary to the main ideas under discussion, and will not be considered any further.

4.9 Management Services

The external-management services provided by the construction facility should indicate, for any particular object, whether a specified destination-type can be generated. Also, there is a requirement to provide information on the 'cost' of a construction, thereby allowing the scheduler to compare different options. Comparisons are also required when more than one source-type is available for an object, resulting in multiple construction paths leading to the same destination. This situation can occur, for example, if the object has been implemented in more than one language.

The feasibility of a particular construction can be answered simply by searching for a suitable construction path. If at least one path exists then the construction can be performed, otherwise it can not. Comparing different paths requires additional information relating to construction 'costs'. In principle, each arc on the construction graph has an associated cost, as shown in Figure 4–5. The cost of a particular construction is given by the cumulative cost of the individual transformations from which the construction path is composed. Finding the 'cheapest' path is then a special case of the shortest path problem for weighted graphs; a standard problem for which standard solutions exist.

An intuitive cost function is the time taken to perform the transformations. The construction cost is then measured as the cumulative time taken to perform the series of transformations leading to the desired object representation; the path requiring the least time being the best. Unfortunately, the transformation time depends upon many factors, including the 'size' of the source representation to be transformed. Hence, the transformation time will be different for *every* object at *every* transformation step.

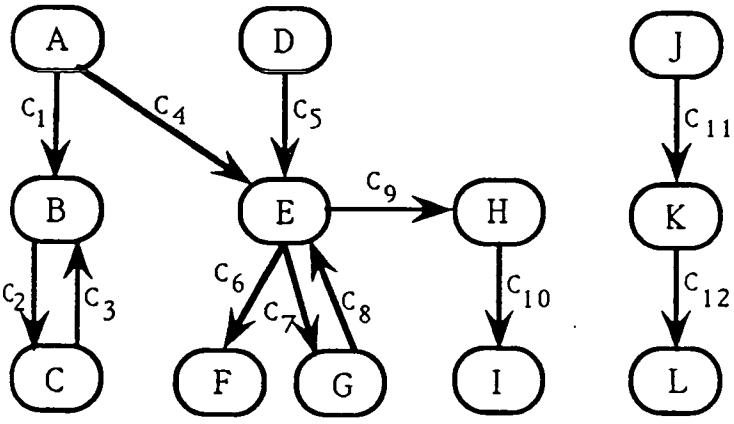

Figure 4–5: A Construction Graph with Cost Labels

If a usable measure of the representation 'size', n, could be specified, then transformation tools could be tagged with their time complexity, for example, $O(n)$, $O(n^2)$ etc., thus yielding an estimate of the transformation time. However, even if such measures were feasible, other 'difficult-to-measure' factors such as current system performance must also be taken into account. In reality, accurately predicting transformation time is dependent upon too many unquantifiable variables. Hence, the conclusion is drawn that, although desirable, transformation time is not a practical measure of construction cost.

Another approach is to try and examine the system resources used to perform transformations; the path using the 'least' resources being the best. Measuring resource usage is discussed extensively in chapter 5 in relation to scheduling. Without pre-empting that discussion, it is safe to state that measuring resource usage suffers from the same quantification difficulties as predicting transformation time, making it equally unusable.

The problem of defining a suitable cost indicator is similar to that of defining 'workload' in distributed scheduling (see chapter 5). An approach often adopted in scheduling is 'simple is best'. This is based on the experience that the benefits

of using very accurate, complicated metrics, are outweighed by the effort required to observe and update them.

The proposed implementation described below includes the simple measure of path length, i.e., the number of transformations required between source and destination types. Path length has the advantage of being both easily calculated and interpreted, but suffers from the obvious disadvantage of being very coarse. Using this measure, the shortest path is considered best, regardless of the individual transformations involved. The algorithm shown does not depend upon the use of a path length cost indicator, so other metrics could used within the same framework.

4.10 A Proposed Implementation

This section describes a design for a distributed object construction service based upon the construction graph concept described above. A distributed data structure is defined that embodies the construction graph for a complete distributed environment. Construction paths are then found by applying standard graph searching algorithms to this distributed graph structure.

4.10.1 Type Representatives

The design is based upon defining a **Type Representative**[2] object. Each transformation node in the construction graph is realised as an instance of object `TypeRepresentative`. For example, the Simula representative embodies all knowledge relating to transformations of Simula objects. Similarly there will be a C++

[2]The term 'representative' is loosely derived from the phrase 'representation service'.

Transformation Tool	Output	
	Type	Representative
T ↦ U compiler	U	U_Rep
T ↦ V compiler	V	V_Rep
T ↦ S translator	S	S_Rep
T ↦ W compiler	W	W_Rep

Figure 4–6: The T Representative Output Configuration Table

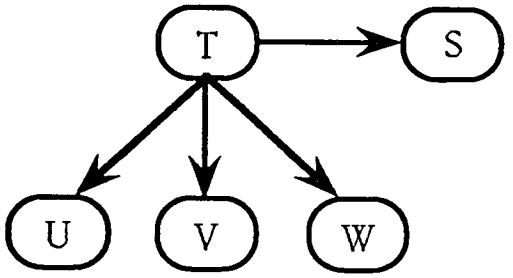

Figure 4–7: The T Representative Output Graph Segment

representative, an assembler representative and so forth. The representatives may be fully replicated to provide easy access, but each copy must maintain consistency with the others; logically there is only one representative per representation type. As an example, every host might maintain a local copy of its corresponding representative (e.g., the SUN3-Unix representative).

A representative defines a mapping between its input type and its output types. The outputs are determined by the transformation tools available for manipulating the input type. Each representative therefore keeps a table of applicable transformation tools and their output types. A specimen table for the T representative is shown in Figure 4–6. The corresponding segment of construction graph it defines is shown in Figure 4–7. Each of the associated output types S, U, V and W will also have representatives defining adjacent segments of the construction graph.

Input Representative
U_Rep
P_Rep
Q_Rep
R_Rep

Figure 4–8: The T Representative Input Configuration Table

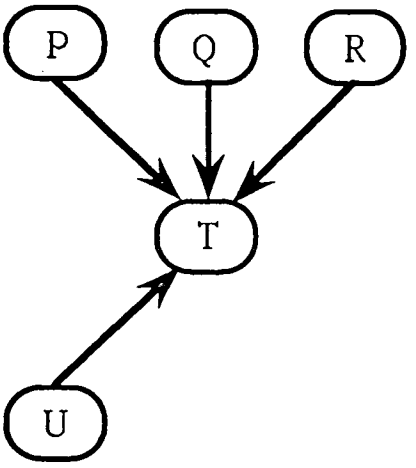

Figure 4–9: The T Representative Input Graph Segment

In order to implement a construction path search, each representative must also keep a list of those representatives capable of producing its input type. The T representative's input configuration table is shown in Figure 4–8, with the corresponding graph segment shown in Figure 4–9. Hence, the segment of construction graph known to the T representative is as depicted in Figure 4–10.

The services offered by a type representative are outlined in the following list. Each service is subsequently discussed with respect to its role in providing an object construction facility.

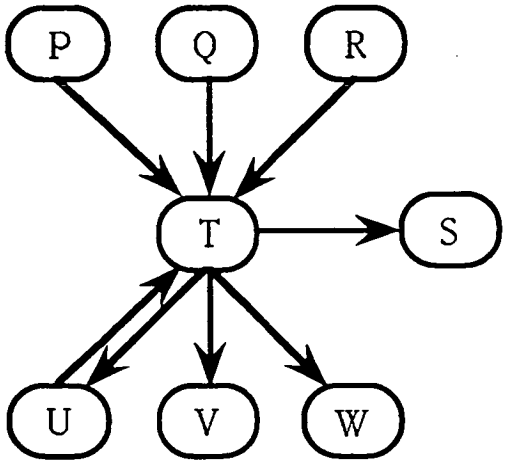

Figure 4–10: Construction Graph Segment Known to the T Representative

- ▷ Maintain a configuration table of possible transformations relating to a specified type.
- ▷ Allow queries and updates on this table.
- ▷ Participate in searching for construction paths
- ▷ Participate in object construction
- ▷ Manage cached representations

4.10.2 Maintaining Configuration Tables

Each type representative must maintain input and output configuration tables as defined above. As the distributed environment evolves, it will be necessary to modify these tables accordingly. For example, if a new T compiler becomes available, an extra entry must be added to the T representative's output configuration table. Similarly, if a new transformation service is added that generates type T as output, it must be added to the T representative's input configuration table. Removal of transformation services must also be reflected by deleting the appropriate

table entries. These modifications to the construction graph are expected to be infrequent events, occurring over days or even weeks, rather than micro-seconds or seconds.

The `TypeRepresentative` object must therefore provide invocation routines such as `AddEntry`, `DeleteEntry` and `EditEntry`. Responsibility for notifying representatives of transformation tool additions and removals is assumed to lie with the (human) system administrator(s), although the representatives can handle the resultant ripple effects themselves. For example, in Figure 4–10, if the R↦T compiler is removed, the system administrators must inform the R representative using `DeleteEntry`. However, the R representative itself can invoke the T representative in order to be removed from T's input configuration table.

4.10.3 Searching For Construction Paths

The search for a construction path is initiated by the object scheduler, either through a `MakeInstance` request, a `CanLocate?` query, or a `WhatCost?` query. In each case the destination location, and hence the destination-type, is specified. This information provides the starting point for the search, namely, the `TypeRepresentative` corresponding to the destination-type.

The algorithm about to be described is a breadth-first graph search. Normally, graph searching algorithms are recursive. However, with the distributed graph structure defined above, each type representative is only aware of a small graph segment, i.e., the part it defines, and hence, is only able to search this small segment. A type representative therefore does not recursively call itself, but rather, calls its peers in adjacent graph segments. Hence, the 'recursion' in the algorithm is distributed between the type representatives. In principle therefore, the graph search can be performed in parallel.

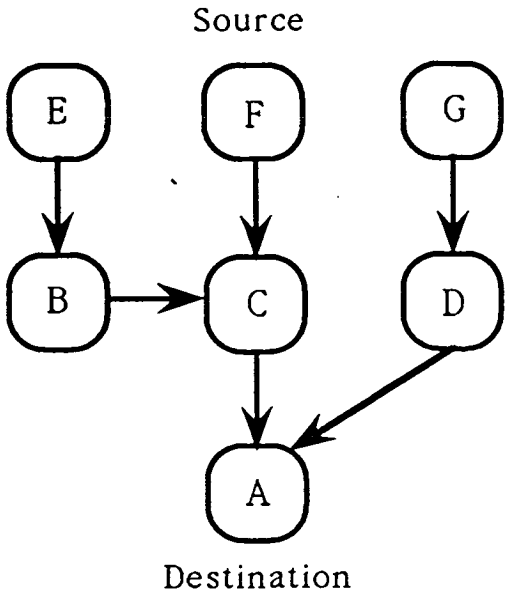

Figure 4–11: An Example Graph : To be Searched

Each `TypeRepresentative` instance has two invocation routines used to perform graph searching :

```
Search(SearchID, ObjectName)
SearchComplete(SearchID, OutCome, PathLength)
```

Two routines are required because the algorithm assumes asynchronous invocation. An algorithm based upon synchronous invocation would only require one routine, but would not be able to handle more than one search at a time or perform individual searches in parallel. The use of these routines will be explained with reference to the graph in Figure 4–11. The type representatives A to G are assumed to have correct input and output configuration tables to describe this graph. The required host-type is A, and the only available source is of type F. Note that, for the moment, the complications introduced by cyclic graphs are ignored.

A mock implementation of `Search()` is shown in Figure 4–12. The `SearchID` is a unique label identifying the current search, and `ObjectName` is the name of the

```
Search(SearchID   : UniqueIdentifier;
       ObjectName : AnyObject       );

     BEGIN
     IF { cached representation available }
          THEN SearchComplete(SearchID, Successful, PathLength = 0)
          ELSE BEGIN
               IF { input configuration table not empty }
                    THEN BEGIN
                         FOR { each peer representative in input table }
                              DO BEGIN
                              Representative.Search(SearchID, ObjectName);

                              { Record search in progress by Representative }

                              END of FOR;
                         END of THEN
                    ELSE SearchComplete(SearchID, Failed);
               END of first ELSE;
   END of Search;
```

Figure 4–12: A Distributed `Search()` Implementation

object to be constructed. The name of the object invoking the search operation is also required, but this is assumed to be provided automatically by the invocation mechanism and is therefore not included as an explicit parameter.

The representative at the root of the search (A in the example) checks to see if it already has a cached copy of the desired object. If no source representation is available (as in the example), then the search is propagated to each of the representatives in the input configuration table (B, C and D) by 'recursively' invoking `Search` on each of them. As explained earlier, these are not recursive invocations upon (A) itself, but invocations upon its peer type-representatives (B, C and D) that may proceed in parallel. Each representative thus invoked becomes the root of a new sub-search. The originating representative (A) records the details of each sub-search initiated, for use later by the `SearchComplete` routine.

In the example, the B, C and D representatives also fail to find a source rep-

resentation, and therefore propagate the search to E, F and G respectively. The E-representative and G-representative will both fail completely because they have neither a source representation, nor any input representatives to further propagate the search. They therefore invoke `SearchComplete` on B and D respectively, with the outcome `Failed`. The actions of `SearchComplete` will be examined shortly. Finally, the F-representative *does* have access to an F-representation of the specified object and therefore reports `SearchComplete`, with outcome `Success`, to the C-representative. The path length, set to zero at this point by the F-representative, will be incremented by each representative as the path information travels back towards the original root.

In the simple algorithm presented here, each sub-search is completed either by failing, or finding a source. No attempt is made to terminate ongoing (parallel) sub-searches once a source has been found. In general this is what is required, since there may be more than one source available and the first one found is not necessarily the 'best'; this depends upon the nature of the cost indicator. With the simple path length indicator used here, the first source found will in fact also be the best. In this case a successful `Search` could broadcast a `StopSearching(SearchID)` invocation in order to terminate any active (now redundant) sub-searches.

The corresponding mock implementation of `SearchComplete` is shown in Figure 4–13. As with `Search` there is a further, implicit parameter included, i.e., the name of the representative reporting the search completion. This name, together with the `SearchID` is used to note the `OutCome` of the sub-search against the information recorded when it was instigated. The representative then checks to see if all its sub-searches have concluded. If not, then no further action is taken. However, when the final sub-search is completed, the results are examined for a `Successful` outcome. If there are none, then a `SearchComplete`, with `OutCome` `Failed`, will be passed further down the line towards the root. For example, the B-representative will receive a `Failed` message from the E-representative. As this

```
SearchComplete(SearchID   : UniqueIdentifier;
               OutCome    : Successful or Failed;
               PathLength : Integer             );

     BEGIN
     { Record OutCome against the information }
     { stored when Search() was invoked       }

     IF { All searches complete }
          THEN IF { none successful }
               THEN SearchComplete(SearchID, Failed)
               ELSE BEGIN
                    { select successful reply with shortest path }
                    { (throwing away all the others)             }

                    { increment PathLength by one }

                    SearchComplete(SearchID, Successful, NewPathLength);

                    END of ELSE;

     END of SearchComplete;
```

Figure 4–13: A Distributed `SearchComplete()` Implementation

was the only sub-search it instigated, the B-representative examines the result and reports the failure to the C-representative. The C-representative however, must wait for a result from the F-representative before passing on any further information. When this `Successful` result arrives, the C-representative will invoke `SearchCompleted` on the A-representative with outcome `Successful` and `PathLength` one.

If more than one source is found, then one of them must be selected. For example, if sources were available to both the E-representative and the F-representative, then the C-representative will receive two `Successful` results. One of these will have a `PathLength` of two, corresponding to the sub-path EBC, and the other will have a `PathLength` of one, corresponding to the path FC. The C-representative ignores the longer path, reporting only the shorter of the two to the A-representative. The same mechanism can be used if there are three or more `Successful` sub-searches. In the case where more than one path corresponds to the shortest length, random selection can be used. The type representatives are expected to retain the information relating to successful searches, for use in subsequent construction requests.

When the graph search concludes, the initiating type-representative, A, has available the information required to satisfy the object scheduler's original request. The result of a `CanLocate?` query depends upon whether or not a path was found; in the example, the result is `Yes`. A `WhatCost?` query receives, in this case, the response `PathLength` 2. The path length metric can be replaced by other cost functions subject to a decision mechanism being available to select the 'best' path at each step. Finally, with regard to a `MakeInstance` request, each representative on the successful path (F, C and A) has retained the name of its 'supplier' for use with the construction algorithm described later.

The search algorithm presented above describes a basic construction service, demonstrating the basic principles involved. There is considerable scope within a

real implementation for improving efficiency, such as using 'hints' that restrict sub-searches to paths more likely to produce a successful result. Hints may be derived by the system through analysis of previous searches, or they may be provided as `Options` parameters to the `Search` routine; perhaps specifying precisely the path to be used. A historical record of successful paths could also be retained to avoid repeated searching.

4.10.4 Performing Transformations

Having found the best construction path for an object, the destination type-representative may now initiate its creation. This can be performed using the `Construct()` routine shown in Figure 4–14. Each representative supervises the application of the transformation tool appropriate to its section of the construction path. If the input representation is not immediately available, then the `SearchID` is used to index the information stored when the construction path was found. This provides the name of the next representative along the construction path, which is then instructed to create the appropriate input representation by a 'recursive' call on `Construct()`. Note that, as with the search algorithm, this is not a recursive call on itself, but a call to a peer routine further along the construction path. Hence, in the example, the A-representative calls upon the C-representative, which in turn calls upon the F-representative. The F-representative applies the F↦C transformer, passing the resultant C representation to the C-representative, which in turn applies the C↦A transformer, thereby producing the required representation.

4.10.5 Managing Cached Representations

The mechanisms used to manage the storage of object representations are not addressed by this thesis. It envisaged that suitable facilities for representation

```
FUNCTION Construct(SearchID : UniqueIdentifier) : ObjectReference;

     BEGIN
     IF { cached representation available }
          THEN Input := SourceObject
          ELSE Input := SourceRepresentative.Construct(SearchID);

     { apply transformation tool to Input, producing Output }

     Return(Output);

     END of Construct;
```

Figure 4–14: A Recursive `Construct()` Implementation

storage and retrieval could be provided using standard database technology, and hence are within the capabilities of the envisaged target environment. It is therefore assumed that each `TypeRepresentative` can, if it chooses, reliably store the output of any transformations it initiates. It is further assumed that the existence of a cached representation can be easily verified and, if required, easily retrieved. Responsibility for implementing the cacheing policy lies with the type representatives.

4.10.6 Searching Cyclic Graphs

The construction graph shown in Figure 4–11 contains no closed loops, i.e., it is an acyclic graph. This section examines the algorithm's ability to search cyclic construction graphs, i.e., those containing inter-related representations.

Figure 4–15 shows an example construction graph containing a cycle. It is identical to the example used earlier except for an additional arc between representatives A and C, corresponding to an A$\mapsto$C transformation tool. Thus, it becomes possible to take an A-representation of an object, using it to produce a C-representation, which in turn can be used to produce an A-representation.

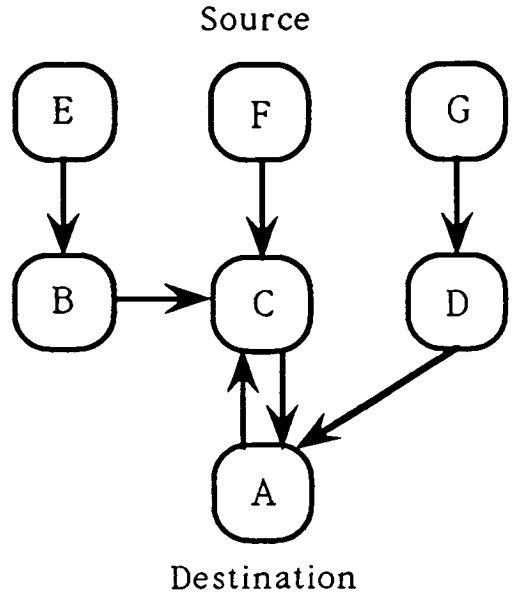

Figure 4–15: An Example Cyclic Construction Graph

This is the simplest form of cyclic loop. Longer loops, containing three or more representation are also possible. The search algorithm described above will enter an infinite loop with such a graph, with the A- and C-representatives continually forwarding the search to each other.

Infinite looping can be avoided by marking each node in the graph as it is visited. This is a standard technique employed in graph searching algorithms. If the search encounters a node it has already visited, then a cycle must exist. Having detected a cycle, appropriate action can be taken to avoid infinite looping.

In the search algorithm described earlier, each type representative records all sub-searches it initiates. This effectively acts as a 'marker' indicating that the search has reached this representative. If a type representative observes a repeat search request, then (assuming no errors) this implies that the search has been propagated around a cyclic path and, furthermore, that no source was found along that path (otherwise the search would have terminated). Any type representative observing a repeated request should therefore invoke `SearchComplete()` with the

outcome `Failed`. This will propagate back around the loop until either it meets a `Successful` result, in which case it is rejected in favour of the successful path, or it reaches the original root of the search. In both cases an infinite loop is avoided.

4.11 Summary

It has been argued that the automatic manipulation of object representations is both desirable and necessary in a heterogeneous environment. A control mechanism for applying transformation tools has been developed using a graph-based formulation of the problem. Standard graph searching techniques, in particular breadth-first searching, have been applied to the distributed graph structure in order to find a source representation for a specified object. The graph search also establishes the transformation steps necessary to create the desired representation. Having found this information, constructing the object becomes straight-forward. The search and construction algorithms presented provide the basis for a working implementation, suggesting that an automatic construction service is a feasible proposition.

Chapter 5

Distributed Scheduling

This chapter provides an introduction to resource scheduling. The scheduling problem is defined in general terms, followed by an overview of current, non-object scheduling techniques. The problems of performance measures and status update policies are also covered. Scheduling in object based systems is then examined, introducing an additional, fine grained level of scheduling; invocation scheduling. Several of the scheduling techniques identified earlier in the chapter are then re-examined with respect to invocation scheduling. It is argued that object oriented performance metrics are more appropriate to an object based environment than the re-application of process based metrics. Several object oriented metrics are suggested as being suitable for performance measurement.

5.1 Introduction

Scheduling is the management of consumer access to resources. The **Scheduling Policy** defines the criteria against which this access is governed. There are many different formulations of this problem depending upon the definitions of consumers and resources. Within the target environment, all **Resources** are objects and, conversely, all objects are resources. No other form of resource exists. The resource **Consumers** are also objects, making invocation requests to satisfy user demands.

The demands made upon a distributed system by consumers represent its **Workload**. In the target environment the workload corresponds to invocation requests. **Distributed Scheduling** (scheduling 'in the large') is the global assignment of resources to workload, governed by the scheduling policy. **Local Scheduling** (scheduling 'in the small') is the assignment of local resources to individual tasks within the confines of a single object host. The problem of local scheduling in single-processor systems has been thoroughly researched, with standard techniques being presented in most basic operating system texts. Local scheduling is therefore not addressed here.

5.2 Scheduling Policies

Many different approaches to distributed scheduling have been suggested in the literature. The plethora of policies have prompted at least two attempts to create a **Scheduling Taxonomy**, i.e., a descriptive framework with which to classify different policies. One of these, by Wang and Morris, is based upon the level of information required by the policy [Wang 85]. Seven levels of information dependency are identified, which are then used to classify the 10 'canonical' scheduling algorithms identified by their study. A more descriptive taxonomy is provided in [Casav88],

which defines a hierarchical classification scheme based on policy characteristics. In describing this taxonomy, Casavant and Kuhl present a comprehensive overview of distributed resource scheduling, including an extensive literature survey classifying over fifty published algorithms.

The following sections provide an overview of basic scheduling techniques and terminology[1]. The techniques identified are not entirely independent. Many scheduling mechanisms exhibit characteristics from more than one 'category'. The majority of distributed scheduling research relates to non-object environments. Schedulers in these systems deal exclusively with *processes*, i.e., executable or executing (binary) program representations, and *processors*, i.e., hardware capable of executing processes. The scheduling problem is therefore defined in terms of assigning processes to processors. This phraseology will be used for the moment. The effect of considering an object oriented environment will be examined later.

Load Sharing and Load Balancing

Load Sharing scheduling policies attempt to distribute the system workload between processors. **Load Balancing** is a specific case of load sharing in which the intention is to keep all processors equally utilised (in some sense), avoiding the situation where one processor is overloaded while, at the same time, another is underloaded. The motivation behind load balancing is the belief that by spreading the workload as evenly as possible, the average completion time per unit of work will be minimised.

In order to maintain this equilibrium, load balancing policies migrate work away from 'overloaded' hosts towards 'underloaded' hosts, thereby correcting any

---

[1] As in chapter 1 with the phrases 'distributed system' and 'object orientation', these terms are not universal.

imbalances that arise. A load imbalance may occur, for example, when previously scheduled work is completed, freeing the resources that were allocated to it. Load balancing relies on the ability to migrate processes in mid-execution, which is an 'expensive' operation because of the need to transfer the current execution state of the process. This can become even more expensive in an heterogeneous environment where the originating and destination processors may require different representations of this state. A survey of the process migration mechanisms[2] used in LOCUS, DEMOS/MP, XOS, V and MOS is presented in [Smith88].

Sender and Receiver Initiated

In sender-initiated scheduling policies, congested processors search for processors with light load in order to transfer work. For example, if an overloaded processor is requested to execute an additional process, it may search for an underloaded processor more capable of satisfying the request. Conversely, receiver-initiated policies place the onus on underloaded processors to look for work by canvassing the overloaded processors. Receiver-initiated schemes suffer from the same mid-execution migration expense as load balancing. This occurs because the offers of assistance arrive at the overloaded processors asynchronously with respect to the creation of new processes. Hence, in order to transfer some of the workload, an existing process must be migrated. With sender-initiated policies, the overloaded processor simply forwards the newly arrived request to *start* executing a process; there is no need to transfer any execution state.

A detailed comparison of sender and receiver initiated schemes is presented in [Eager85]. It is shown that if the transfer costs in each case are the same, then sender-initiated performs better at low to middle loadings, while receiver-initiated

[2]None of the systems surveyed in [Smith88] are object based.

performs better at higher loads. This makes sense intuitively, since at higher workloads the burden of scheduling is removed from the overloaded processors, and placed with the underloaded processors, i.e., with those that have the spare processing capacity to cope with it. However, in most systems that implement dynamic migration, the transfer costs are significantly greater for receiver-initiated schemes due to the mid-execution migration expense they incur. Under these circumstances sender-initiated policies give better average performance across all loadings.

Static and Dynamic

The difference between static and dynamic scheduling relates to the time at which scheduling decisions are made. In **Static Scheduling**, the scheduling decision is made at construction time, i.e., when the executable representation is created. Once made, this assignment is never changed. Hence, every time the process is submitted for execution it is always assigned to the same processor. Static scheduling policies assume a stable environment in which the system status observed at construction time, is the same as that observed each time the process is executed.

Dynamic Scheduling policies remove the assumption of a stable, predictable environment and take the more realistic view that little *a priori* status information is available. The assignment of process to processor is delayed until immediately before the process is executed, incorporating the system's current status into the decision making. Each execution of a process may therefore take place on a different processor, reflecting changes in loading status. Hence, dynamic schedulers can potentially adapt to and exploit, changes in processor availability.

Adaptive Scheduling

An **Adaptive** scheduler is one in which the algorithms and parameters used to implement the scheduling policy, change dynamically according to the previous and current behaviour of the system (which in turn was influenced by earlier policy decisions). Note that the algorithms and parameters themselves must change over time for a scheduler to be classed as adaptive. A scheduler in which only the *values* of observed parameters change is *dynamic*, as described above.

One example of an adaptive policy is an algorithm that attaches weights to each of its observed parameters, with the weights being re-calculated in response to earlier scheduling decisions. Therefore, a parameter that was important in earlier decisions, i.e., one that had a large weighting, may become insignificant in future decisions due to its weighting being reduced. Schedulers based on stochastic learning automata, such as the one reported in [Mirch86], provide further examples of adaptive scheduling.

Probabilistic

Probabilistic scheduling policies are based on the long term probabilistic behaviour of a system. Assignments of processes to processors are performed randomly according to some distribution that describes this behaviour. One of the simplest policies is to assign uniformly at random, for example, in a system of n processors, each is selected with probability $\frac{1}{n}$. In theory, the long term behaviour of such a policy will assign work evenly between all n processors. A more complicated algorithm might adjust the selection probabilities depending upon the observed 'performance' of earlier assignments; a good performance leading to an increased probability of re-selection.

One-Time and Dynamic Assignment

With **One-Time Assignment**, a process runs to completion on its selected processor regardless of subsequent changes in the load distribution. Schedulers employing **Dynamic Assignment** may re-evaluate earlier scheduling decisions in the light of new or more up-to-date information, possibly leading to the mid-execution migration of a process. For example, the essential difference between load sharing and load balancing described above is that load *balancing* uses dynamic assignment, whereas load *sharing* uses one-time assignment. Note that it is still possible to have a policy employing dynamic assignment whose goal is something other than load balancing, for example, re-assigning a process because of the imminent failure or close-down of its current processor.

5.3 Scheduling Metrics

All but the simplest scheduling policies rely upon monitoring certain system parameters, using the observed values to drive the scheduling algorithm. There are two aspects to this monitoring operation; deciding which system parameters provide useful scheduling information, and deciding how to propagate the observed values throughout the system.

5.3.1 Performance Measures

Many scheduling algorithms presented in the literature assume the parameter values they require are known in advance, i.e., before the process is scheduled. For example, some of the earliest work on load sharing [Stone77] assumed that process execution times and levels of inter-process communication were pre-determined. Some algorithms assume that very detailed scheduling information is available. For example, the algorithm presented in [Chou 82] uses :

- ▷ $E(i,x)$: execution time of task i on processor x
- ▷ $CO(i,x,j,y)$: communication time for the results of task i on processor x to task j on processor y
- ▷ $F(i,x)$: probability task i fails on processor x
- ▷ $CH(i,x)$: time to create a checkpoint for task i on processor x
- ▷ $RE(i,x)$: time to restart failed task i on processor x
- ▷ $CI(\alpha,x)$: time to initiate a set of concurrent tasks α by processor x
- ▷ $CC(\alpha,j,x)$: communication time for the results of a set of concurrent tasks α to task j on processor x

The algorithm presented in [Hsu 86] assumes, among other things, that the amount of unfinished work per host is known. The algorithm in [Varad88] includes parameters such as the fixed cost of migrating one unit of resource, and the average resource requirements for a job.

Using this approach the scheduling problem is formulated as an equation, with a parameter included for every characteristic the scheduler's developers believe to be important. The equation is then solved algebraically, the result forming the basis of a scheduling algorithm. If an exact solution is found, then the resultant algorithm is optimal for the problem it describes. Simulations are often used to confirm this. In order to apply an algorithm to a specific system, the parameters are interpreted in units suitable to that system. However, in practice some of these parameters are somewhat difficult to quantify. For example, how is 'one unit of resource' actually defined?

Without exception, all distributed schedulers implemented in working systems use much simpler measures. System developers have taken the pragmatic approach of using parameters that are available and easily measured. There are

principally two metrics in common use; processor **Queue Length** and percentage processor **Utilisation**. Queue length measures the number of processes currently active on a processor, thus providing a coarse approximation to the processor's load. Processor utilisation measures the percentage of time the processor is active (or equivalently the percentage idle time), thus yielding a different approximation to processor load. Queue length is used, in conjunction with utilisation, by the LOCUS distributed file system [Hać 86]. Utilisation is used within the MOS distributed system [Barak85] and the V distributed system [Theim86].

Queue length does not provide the same level of accuracy as utilisation, since an interactive program such as a text editor, does not present the same processor load as a computationally intensive program, such as a simulation. Both programs, however, carry equal weight when measured as entries in the processor queue. Utilisation, although not as coarse, requires greater effort to determine, since it is usually calculated as an average over some specified time period in order to smooth temporary fluctuations.

Simple scheduling mechanisms are employed principally to reduce implementation complexity. However, a study by Eager *et al.* has shown that :

> "Extremely simple load sharing policies using small amounts of information perform quite well—dramatically better than when no load sharing is performed, and nearly as well as more complex policies that utilize more information." [Eager86]

The study's conclusion is that the expense of propagating and maintaining complex status information, is not justified by the marginal improvements gained in scheduling performance, i.e., "simple is best".

5.3.2 Status Update Policies

Given a usable performance indicator for a processor, it becomes necessary to inform other processors of its current value, keeping them up-to-date with any changes. There are many possible update policies, the simplest of which is **Periodic Broadcasts**. Under this policy, each processor observes its own performance metric every T seconds, broadcasting the new value to all others. Hence, each processor has an estimate of the others' current performance, guaranteed to be no more than T seconds out of date. There is a tradeoff here between update overheads and accuracy, determined by the timeout value T. A small value for T generates more updates and therefore greater accuracy, but incurs greater maintenance overheads. Conversely, a large value of T reduces the overheads, but also reduces accuracy.

A refinement to periodic broadcasting, attempting to reduce the number of update messages while still retaining accuracy, is suggested by Theimer who observes that, with a sender initiated policy, the only updates of interest are those from lightly loaded processors [Theim86]. He therefore suggests a scheme in which a cutoff level is defined, such that only processors whose load is below the cutoff level send updates, those above the cutoff level remaining silent. The cutoff level can be modified over time and re-broadcast to reflect changes in the overall system load. This mechanism has been implemented in the V system.

Probing and bidding policies provide examples of update mechanisms that propagate status information only when it is requested, rather than supplying continuous, unsolicited updates. With **Probing** [Eager86], when an overloaded processor receives an 'execute new process' request, it randomly selects one of the other processors and 'probes' it, i.e., sends a status request message, to establish its current load. If adding the new process to the probed processor would not make it overloaded, then the 'execute' request is transferred. If the probed processor is already overloaded, or if adding the new process would make it overloaded, then a second processor is randomly selected and probed. This continues until either

a suitable processor is found, or the number of processor probed reaches a predetermined limit. In the latter case the originating processor must execute the process itself. With a **Bidding** algorithm, a processor wishing to offload an 'execute' request sends a broadcast message giving details of the process. Any processor that wishes may then respond with a 'tender' for the process. The responses are analysed, with the 'best' tender being awarded the new process.

Another mechanism, reported in [Ni 85], attempts to reduce update traffic by only sending updates when there is something worth reporting, i.e., when there has been a notable change in a processor's load. In order to define 'a notable change', three system-wide loading categories are used; high load, normal load and light load. Each processor is assumed able to classify its current load in terms of these categories. An update message is only broadcast when the processor's load crosses a category boundary. Unfortunately, this scheme suffers from what the authors call 'state-woggling', which occurs when a processor's load lies on a boundary, continually fluctuating between two categories and thus, continually generating updates. The idea of only reporting 'notable changes' will be returned to later in the thesis.

5.4 Scheduling in Object-Based Systems

The systems mentioned above in relation to process migration (LOCUS, DEMOS/MP, XOS, V and MOS) are all process based, i.e., not object oriented. Most current object oriented systems providing *object* migration use exactly the same techniques as their process based counterparts. For example, the migration mechanisms of Amoeba and Eden treat migratory objects simply as executable process images. One notable exception is Emerald in which *all* objects can move regardless of size [Jul 88], including simple data objects.

The remainder of this chapter examines the effect upon scheduling of employing object oriented characteristics, rather than reducing everything to the level of binary processes. Several of the topics introduced earlier are re-examined in the light of this new perspective.

Definitions

In chapter 2, the Object Reference Model identified two tiers of scheduling in object based systems; object scheduling and invocation scheduling. **Object Scheduling** is the assignment of object instances to object hosts. This is comparable to the assignment of processes to processors in process based systems. **Invocation Scheduling** is the assignment of invocation messages to object instances. There is no real parallel here with process based systems. The need for invocation scheduling is the principal difference between objects and processes, a distinction that raises some interesting scheduling possibilities.

The invocation scheduler has two types of objects to consider; those whose service is **Immutable**, and those whose service is **Retentive**. Invocations upon an immutable service are independent and can be passed to any instance offering the service. For example, the result of invoking routine `Multiply` will be the same regardless of the `Calculator` instance used. Invocations upon retentive services however, are related and must always be passed to the same instance. Retentive services result when an invocation causes the object to alter its internal state, thereby 'remembering' that the invocation occurred. This generally occurs with transaction based services that have an initialisation phase, followed by a period of service provision, followed by a closedown phase. As an example, invocations on `EnQueue` and `DeQueue` must be passed to the same `Queue` instance in order to have the desired effect.

Load Balancing

In process based systems, load balancing is synonymous with process migration. This results from that fact that the smallest unit of workload is a complete process. Migrating work to balance the load therefore implies migrating the process representing the load. In an object environment this equates to the migration of object instances between hosts; the principal load balancing technique employed by current object oriented systems. However, in object based systems the smallest unit of workload is represented not by objects, but by invocation messages. Indeed, objects by themselves do not usually represent *any* load without invocation messages to request their services. The granularity of workload as measured by invocation messages, is therefore much finer than in process based systems; closer to the 'procedural' level than the process level. Since invocation messages can be migrated around the system with relative ease (minimal expense), object oriented systems offer an additional, finer grained level of load balancing by scheduling invocations between replicated servers.

The simplest case is that of an immutable service. When the arrival rate of invocations exceeds an immutable object's servicing capacity, an additional instance can be created to handle the overspill. Hence, the *load* has been migrated without migrating objects. As long as there is spare capacity in the system to host additional instances, an immutable service can be 'expanded' in this manner to match the current demand. A corresponding 'garbage collection' mechanism should also be employed, removing redundant instances during periods of low invocation activity.

Expansion of retentive services is not as straightforward because of the dependencies between consecutive invocations. However, there are still benefits to be gained in taking this approach. A retentive service can be expanded subject to the restriction that the new instance only handles *new* transactions, i.e., those for which the original instance holds no status information. Hence, the additional

instance cannot alleviate the original's load, but it can prevent the load from increasing further. An improvement upon this scheme would be to allow the transfer of *some* status information between instances, via 'transaction-transfer' invocation routines. This would allow the original instance to transfer enough of its current 'transactions' to balance the load. This is not the same as transferring a process's context information, which is an 'all or nothing' machine — and implementation — dependent transfer of binary data. 'Transaction transfers' would enable an object to off-load as much, or as little of its workload as required.

Service specific status information should be passed as implementation independent, structured data types, rather than as binary images. The transfer could use the standard invocation mechanism to invoke routines in the recipient object specifically provided for this purpose. One immediate and very important consequence of transferring status at the object level, rather than the process level, is that it transcends both host heterogeneity and implementation heterogeneity. The original and new instances can communicate service specific information, even though they may be heterogeneous implementations residing on heterogeneous hosts. The language CLU implements a value transmission method similar in spirit to that required here [Herli82].

The conclusion is that in object based systems, the fine-grained workload allows load balancing to be implemented without the overheads of object migration. If status information needs to be transferred with the migrated work, it should be passed via invocations as high level service specific data types in order to avoid problems with system heterogeneity. This philosophy is summarised by the phrase "Don't migrate, replicate!".

Performance Measures

In an object oriented environment, the analogous load measure to queue length is **Object Count**, i.e., the number of objects present on a host. Object count suffers from the same problem as queue length in that not all objects represent the same level of workload; an object never invoked generates little or no load. Further, object count gives no information regarding the load on each individual object, which is a necessary prerequisite for invocation scheduling. Using host utilisation as a load measure suffers from the same drawback, in that it provides no object related workload information. The standard workload measures used to implement scheduling in process based systems therefore do not transfer easily into an object oriented environment; something else is required.

The following paragraphs discuss the monitoring of object performance for use in conjunction with invocation scheduling. Since hosts are themselves objects, it is possible to view object scheduling as a special case of invocation scheduling. Therefore, the comments made below are generally also applicable to object scheduling. It has already been argued that, at least with respect to load balancing, considering object oriented characteristics yields an improvement over the process oriented approach. This approach will also be applied here to derive an object oriented performance metric.

There are two basic methods of defining object oriented performance metrics. Either provide every object with its own specific metric, or define generic metrics applicable to all objects. The former offers potential for greater accuracy, while the latter (if possible) would be more easily interpreted, requiring no specific knowledge of the object under observation. These two approaches are discussed below.

If object specific metrics are employed, they must be defined by the object's implementor as part of the implementation. The fact that they are object spe-

cific implies the object must participate in the monitoring activity, for example by providing an additional invocation routine such as `CurrentLoad()`, which calculates and returns the current load in units specific to the object. An invocation scheduler is therefore provided with the means to interrogate object instances, establishing their loads. The major problem with this approach is in interpreting the myriads of different metrics it generates. Different implementors will use different metrics, possibly even when implementing the same object, thus making it impossible to compare their relative performances. Although regulations could be devised to avoid such conflicts, this approach is not considered worthwhile. Another drawback is that it forces every object instance to participate in scheduling. This is not seen as desirable, since scheduling should be a function of the system, and not reliant upon the cooperation of the objects being scheduled or, perhaps more importantly, not reliant upon the object's implementor.

A more manageable approach is to employ generic object attributes possessed by all object instances. There are many such attributes, some of which have potential for use as performance metrics. Examples include : the number of invocation routines; invocation rate (averaged over the previous T seconds); service rate per invocation routine (averaged over the previous n invocations); service rate per object; and many others derived from these, such as throughput (arrival rate divided by service rate). More concrete definitions for some of these metrics will be given in chapter 6.

The fact that these attributes belong to *every* object means they they do not rely upon information specific to a particular object, or any other recondite knowledge. Some of them are **Externally Observable**, i.e., measurable by a third party (such as a scheduler) monitoring the interaction between client and server. If such metrics can be used for scheduling purposes, then this leads to the desirable property of **Passive Participation**, i.e., objects play no part in the monitoring of their own performance.

Chapter 6 now describes a scheduling mechanism based on these ideas, monitoring invocation service times to provide an estimate of object performance. Chapter 7 presents an update algorithm, also based on service times, that eliminates redundant updates, thereby reducing update overheads.

Chapter 6

Comparison Scheduling

A novel approach to invocation scheduling is developed using statistical hypothesis testing as the basis for a scheduling algorithm. The behaviour of two intuitive scheduling policies, random scheduling and greedy scheduling, is examined, with simulated performance results presented. A new scheduling algorithm, known as comparison scheduling, is then developed, based upon hypothesis testing. The comparison scheduler only selects an object if it is significantly better than its peers; otherwise random selection is used. A detailed statistical model is developed to rigorously define the phrase 'significantly better'. The simulated performance results for comparison scheduling are compared to those for random and greedy scheduling. These results show a marked improvement, indicating comparison scheduling's considerable potential for use in object oriented distributed systems. The application of a comparison scheduler to object scheduling is also considered.

6.1 Model of Invocation

Invocation scheduling is the assignment of invocation messages to object instances. The interesting case arises when there are two or more object instances capable of servicing an invocation. The scheduling problem addressed here is how to decide, on the basis of observing only service times, which object instance should receive each invocation. For simplicity, all objects are assumed to provide idempotent services. This means that *every* invocation can be scheduled individually. With retentive services, invocation scheduling is restricted to only the first invocation in a 'transaction', since all invocations in a 'transaction' must be presented to the same instance. (see chapter 5, page 135).

The model of invocation assumed throughout this chapter is shown in Figure 6–1. Each object S provides a service of r_s invocation routines $R_1, R_2, \ldots, R_{r_s}$, collectively denoted R_s. Invocations upon routines in S are placed at the tail of its request queue, which is serviced serially from the front, i.e., in a first-come first-served ordering. Only one routine (the one specified in the invocation message currently being serviced) is active at any one time. Upon completion of each request the object is assumed to send a (potentially empty) reply. This enables an external observer to accurately determine the service time by comparing the arrival and reply times for each invocation request. The observed service time therefore includes the invocation message's queueing time.

Each routine R_i has service rate μ_i, where each μ_i may be different. In order to demonstrate the comparison scheduling mechanism developed later, these service times are assumed to follow an exponential distribution. The use of exponential service times is discussed below.

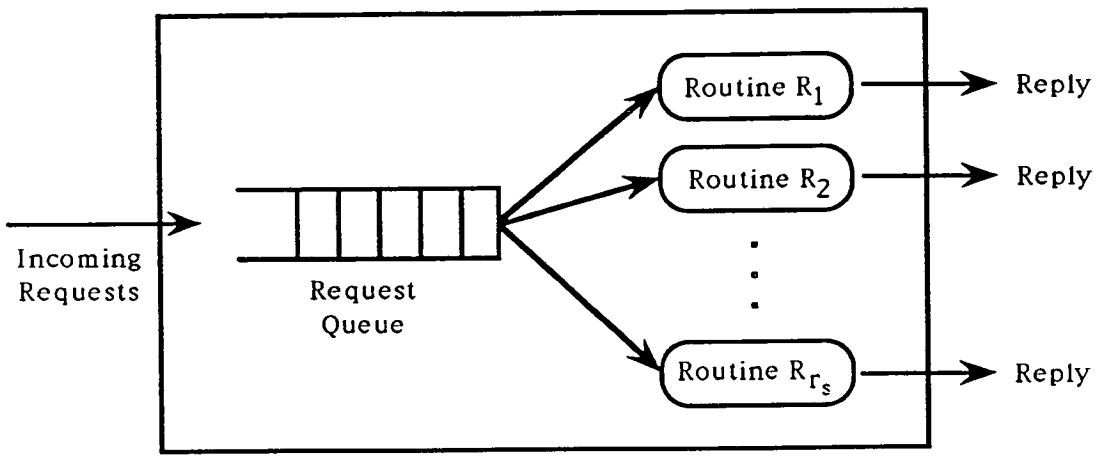

Figure 6–1: Invocation Model

6.2 The Control Scheduling Policies

Before presenting comparison scheduling, two intuitive policies will be examined; random scheduling and greedy scheduling. They are introduced for use as controls with which to compare the new policy.

6.2.1 Random Scheduling

Uniform random scheduling is one of the simplest policies to implement as it requires no status information. Given n suitable object instances, invocations are assigned uniformly at random, each instance being selected with probability $\frac{1}{n}$. The long term behaviour of this policy assigns invocations evenly between the n objects. However, because it does not observe current performance, random scheduling makes no attempt to compensate for any load imbalances that may arise.

6.2.2 Greedy Scheduling

Greedy scheduling uses the average service time of each instance to select the fastest one. The assumption is that recent history on service times provides a reasonable indication of current performance. The term 'greedy' is used because it *always* selects the 'best' i.e., fastest instance. The average service time of each instance is updated upon the completion of every invocation. This average, $\bar{X}_s$, is calculated over the previous h invocations on object S as

$$\bar{X}_s = \frac{1}{h} \sum_{j=1}^{h} T_j$$

where T_j is the observed service time of the j^{th} invocation. When invocation $h+1$ completes, the service time for invocation 1 is replaced by that for invocation $h+1$. Hence, the length of service history maintained remains constant. The simulation results described below compare several different values of h, namely 1, 6, 12, 25 and 50 invocations.

6.3 Arrival Rates and Service Times

The Poisson and Exponential distributions are widely used in simulation work to model arrival and service times respectively. Their principal virtue is the memoryless property (see Poisson postulate three below), which is the key to obtaining analytic solutions to many queuing problems. In practice, rationalising the assumption of Poisson arrivals (and consequently exponential servicing) rests on satisfying the Poisson postulates defined below. For a more detailed discussion on this topic see [MacDo87] (section 1.2) and [Mitra82] (section 4.4).

6.3.1 The Poisson Distribution

The Poisson distribution is commonly used to model the arrival of customers at a service facility, such as the arrival of invocation messages to an invocation scheduler. It is derived from the following postulates (where $N(t)$ is the number of arrivals occurring in time interval t) :

1. In a 'small' time interval of length Δt, the probability of exactly one arrival is proportional to the size of the interval.

   $\text{Prob}[N(t) = 1] = \lambda \Delta t$

2. In this interval Δt, the probability of more than one arrival is negligible.

   $\text{Prob}[N(t) < 1] = o(\Delta t)$

3. The occurrence of an arrival in a small time interval is independent of other arrivals and also independent of the time since the last arrival.

These postulates are assumed to hold in most queuing systems, including the model of invocation used throughout this chapter.

Using only these assumptions, the Poisson distribution function, $P_n(t)$, can be derived, i.e., Prob$[N(t) = n]$.

$$P_n(t) = \frac{e^{-\lambda t}(\lambda t)^n}{n!}$$

For the mathematical details of this derivation see [Maeka87], or any text on queuing theory.

Random variates drawn from the Poisson distribution are said to form a Poisson process or Poisson stream. They model the conditions described by the postulates, i.e., random arrivals with rate λ. The Poisson distribution is used to generate the simulated workload for the scheduling simulation described in this chapter.

6.3.2 The Exponential Distribution

The inter-arrival time between two events in a Poisson stream is the *waiting* time for the second event. Suppose an arrival occurs at time 0. The time of the next arrival is less than or equal to t if and only if at least one arrival occurs in the interval $(0, t)$. The probability distribution of inter-arrival times T is given by :

$$\begin{aligned} F(t) = \text{Prob}[T \leq t] &= \sum_{n=1}^{\infty} P_n(t) \\ &= \sum_{n=0}^{\infty} P_n(t) - P_0(t) \\ &= 1 - e^{-\lambda t} \end{aligned}$$

which is the exponential distribution function, with mean $1/\lambda$.

Hence, the waiting times between events in a Poisson process are distributed exponentially. The exponential distribution is therefore often used to model service times, since a service time can be viewed as the waiting time between starting a service and service completion. The exponential distribution is used to generate invocation service times in the scheduling simulation.

6.4 Simulation Description

A simulation program was implemented to test the performance of random and greedy scheduling. The same program is also used later to test the performance of comparison scheduling. The purpose of this simulation is to establish the *feasibility* (or otherwise) of a scheduler based purely on object oriented attributes. It is *not* intended to define the detailed performance characteristics of a working scheduler. All aspects of the simulation experiments have therefore been kept as simple as possible. In particular, the initial results are based on observing only three object instances. However, as will be demonstrated, the results from this simple configuration are sufficient to expose the deficiencies of random and greedy scheduling, and subsequently the improvements achieved by comparison scheduling.

The three simulated object instances A, B and C, each provide the same idempotent service with five invocation routines, $R_1, R_2, \ldots, R_5$. The choice of five is arbitrary, but not atypical, and was considered the smallest number sufficient to emulate an 'interesting' object. The performance of each scheduling policy (random and greedy) is tested in two separate cases. First, when all three instances offer identical performance, referred to as the **Uniform** case, and second, when their relative performances are uneven or **Non-Uniform**. The purpose of this is to establish how well each scheduler adapts to changes in object performance. In each case the simulation is repeated with three different workload levels denoted low, medium and high. These workloads are defined in terms of the invocation rate upon the three instances. An invocation arrival rate less than the service rate of a single instance is defined as low load, i.e., all invocations are within the capacity of a single instance. Arrival rates exceeding the capacity of a single instance, but less than that of two instances are defined as medium load. Finally, high load is defined as any invocation rate exceeding the capacity of two instances.

Arrival rates exceeding the capacity of all three instances can not be sensibly handled by invocation scheduling alone. It is under these circumstances that service expansion should be used to create an additional service instance. This raises the service capacity to match the workload, whereupon invocation scheduling can re-distribute the load accordingly. Chapter 7 (section 7.4) describes a mechanism for detecting such overload conditions.

The simulated workload was generated independently of the simulator, to be read in during each simulation run. Three workload files were produced, one for each of the load categories low, medium and high. The same files were presented to each simulation experiment so that all scheduling results pertaining to, say, high load correspond to the same (high rate) series of invocations. This enables direct comparisons to be made between the performance results of the different scheduling policies. Each workload file contains a list of start times indicating the points at which each simulated invocation should be generated (Figure 6-2). These times were drawn from a Poisson distribution with arrival rates as defined below. Associated with each start time is a number in the range 1–5, drawn from a uniform distribution, indicating the routine to be invoked. The actual service time corresponding to each invocation is generated during the simulation; drawn from the exponential distribution with a rate dependent upon the routine invoked (see below).

During each simulation run, simple statistics are collected on the scheduler's performance. These are summarised at the end of each run, indicating : the number of invocations simulated; the number assigned to each instance; the average service time of each instance; and the average service time across all instances. The standard deviation of service times, σ_s, is also calculated for each object S as

$$\sigma_s = \sqrt{\frac{\sum_i T_i^2 - \frac{1}{n_s}\left(\sum_i T_i\right)^2}{n_s}}$$

where n_s is the total number of invocations upon object S throughout the simula-

Start Time	Routine
0.1774	3
0.3332	2
0.7900	3
1.2251	4
1.9574	5
2.1056	4
2.5177	3
2.5796	3
3.7144	1
4.0948	2
4.5913	4
4.9182	4
4.9876	3
5.4306	2

Figure 6–2: A Sample of Simulated Workload

tion run, and $T_i 1 \leq i \leq n_s$, is the service time of each of invocation. σ_s is used to provide 95% confidence intervals for the average service times per instance. This interval is calculated as :

$$\left(\bar{X}_s - \frac{1.96\sigma_s}{\sqrt{n_s}}, \bar{X}_s + \frac{1.96\sigma_s}{\sqrt{n_s}}\right)$$

where $\bar{X}_s$ is the average service time for instance S across all invocations. The accuracy of this interval relies on n_s being large enough to allow the use of the Central Limit Theorem, i.e., the interval is based on the Normal distribution rather than the exponential distribution, making it easier to calculate. These confidence intervals provide some idea of the *range* of service times observed throughout the simulation. They therefore indicate the performance *consistency* of an object; the smaller the range, the greater the consistency.

The simulator configuration is shown in Figure 6–3. For the uniform case, i.e., when all three instances offer the same performance, the invocation service rates were fixed as follows : $\mu_1 = 1.0, \mu_2 = 2.0, \mu_3 = 3.0, \mu_4 = 4.0, \mu_5 = 5.0$.

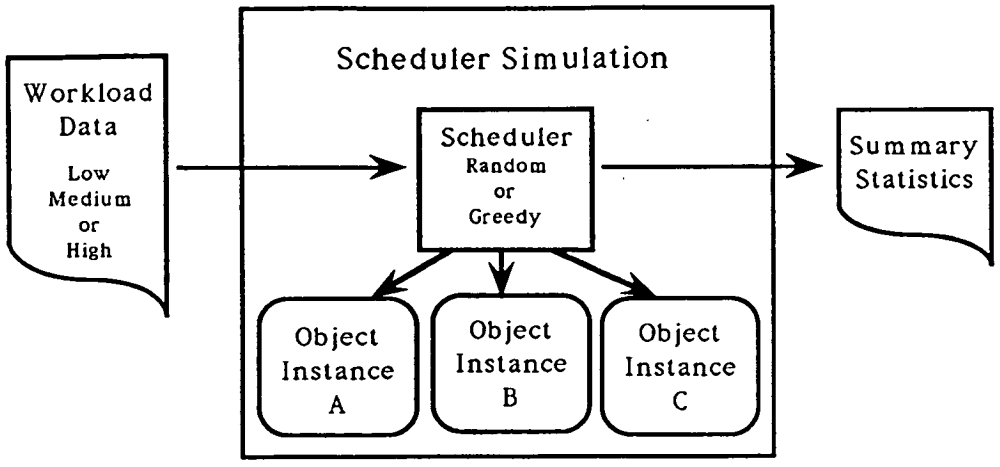

Figure 6–3: Simulation Configuration

Again, these figures (and their units[1]) are arbitrary, however, applied consistently across all experiments they provide a basis with which to compare the *relative* performance of each scheduler. For the non-uniform case, i.e., when each instance offers a different performance, the service rates for instance A are reduced by 50%, the service rates for instance B remain the same, while the rates for instance C are increased by a factor of 50%. The combined service capacity of the three instances therefore remains the same as in the uniform case.

The expected service time for each invocation routine R_i is $\frac{1}{\mu_i}$, hence the expected service time, $\bar{X}$, for each instance is

$$\bar{X} = \frac{1}{5}\sum_{i=1}^{5}\frac{1}{\mu_i}$$

assuming invocations are generated uniformly for the five routines. From the service rates specified above for the uniform case, this yields an average service time per invocation per instance of 0.457. Hence, the effective service rate of each instance, $\bar{X}^{-1}$, is 2.19. Two identical instances therefore offer a theoretical service

[1]The units are notionally seconds, although this is not important

rate of 4.38, with three instances operating at 6.57. Combining these figures with the above definitions for workload, the simulated arrival rates for low, medium and high load are fixed at 2.0, 4.0 and 6.0 respectively. In each case these are realised as the aggregation of five separate invocation streams operating at a fifth of this rate; one for each invocation routine. The same arrival rates (in fact the same workload file) are used in the non-uniform case, allowing the direct comparison of performance results from the uniform and non-uniform simulations.

The workload files generated for each of the categories low, medium and high, simulate invocation arrivals for 3600 units of elapsed time (notionally one hour). Even in the low load case this represents approximately 7200 (3600×2.0) invocations, which is a sufficiently large sample to allow the Normal approximation to be used when calculating the confidence intervals. Each simulation run is in fact performed 10 times to gain a more representative view of each scheduler's characteristics. The results presented in this chapter, except when stated otherwise, are therefore averaged over ten simulation runs.

6.5 Simulation Results

6.5.1 Random Scheduling

The performance results for the random scheduler are shown in Tables 6–1 and 6–2. In the uniform case, random scheduling predictably performs extremely well. The workload is shared evenly between the three instances, with each receiving exactly one third of all invocations. The average service time per invocation increases as the load increases, but not disproportionately. The variation between instances remains very small under all loadings.

In the non-uniform case, the servicing capacity is split 16%, 34%, 50% between A, B and C respectively. However, because random scheduling makes no attempt

Load Category	Object Instance	Invocations Received	Average Service Time
Low	A	33% ± 1%	0.8 ± 0.04
	B	33% ± 1%	0.8 ± 0.04
	C	33% ± 1%	0.8 ± 0.04
	Overall	7344	0.8
Medium	A	33% ± 1%	1.5 ± 0.05
	B	33% ± 1%	1.4 ± 0.05
	C	33% ± 1%	1.4 ± 0.05
	Overall	14358	1.4
High	A	33% ± 1%	6.1 ± 0.1
	B	33% ± 1%	6.5 ± 0.1
	C	33% ± 1%	6.5 ± 0.1
	Overall	21753	6.4

Table 6–1: Random Scheduling Under Uniform Performance

Load Category	Object Instance	Invocations Received	Average Service Time
Low	A	33% ± 1%	3.2 ± 0.2
	B	33% ± 1%	0.7 ± 0.04
	C	33% ± 1%	0.4 ± 0.02
	Overall	7344	1.5
Medium	A	33% ± 1%	414.8 ± 6.7
	B	33% ± 1%	1.3 ± 0.04
	C	33% ± 1%	0.7 ± 0.02
	Overall	14385	139.1
High	A	33% ± 1%	1492.5 ± 17.9
	B	33% ± 1%	6.6 ± 0.2
	C	33% ± 1%	1.0 ± 0.03
	Overall	21753	499.8

Table 6–2: Random Scheduling Under Non-Uniform Performance

to observe this, the invocations are still apportioned evenly. Consequently, the slowest instance, A, is permanently overloaded, while the fastest instance, C, is under-utilised. This is reflected in the large service times for instance A, particularly at medium and high loads, caused by invocation requests spending a large amount of time in A's request queue waiting to be serviced. Consequently, the average response time across all three instances is very poor. Direct comparisons can be made with the uniform instance results since the invocation rates are the same in both cases, as is the combined processing capacity of the three instances.

In conclusion, the simplicity of uniform random scheduling is only beneficial when all object instances offer very similar performance characteristics. When their servicing capacities differ, random scheduling still assigns invocations evenly, leading to a load imbalance and consequently poor average performance.

6.5.2 Greedy scheduling

Before examining greedy scheduling in detail it is necessary to establish the history length, h, to be used. The summarised performance results using different length histories are shown in Tables 6–3 and 6–4. Each entry is the result of only a single simulation run. The service times are considerably worse than random scheduling in almost all cases. Only with low load and non-uniform performance does greedy scheduling outperform random scheduling. This observation holds for all the history levels tested, with no particular history being appreciably better than the others.

The history used in the more detailed analysis has been fixed at 12 invocations. This allows direct contrasts to be made later with comparison scheduling, which also uses a history of 12 invocations (the reason for this will be explained later). The more detailed greedy scheduling performance results, incorporating a history

Load Category	Average Service Time at History Length :				
	1	6	12	25	50
Low	3.5	4.5	4.3	4.7	4.2
Medium	204.7	221.2	246.8	205.5	210.6
High	795.6	696.7	756.7	778.7	628.7

Table 6–3: Greedy Scheduling Under Uniform Performance

Load Category	Average Service Time at History Length :				
	1	6	12	25	50
Low	1.0	1.1	1.2	1.1	1.1
Medium	98.9	99.5	92.5	136.1	100.6
High	605.8	508.2	535.7	542.5	505.4

Table 6–4: Greedy Scheduling Under Non-Uniform Performance

Load Category	Object Instance	Invocations Received	Average Service Time
Low	A	31% ± 23%	4.6 ± 0.2
	B	30% ± 27%	4.9 ± 0.2
	C	41% ± 26%	4.6 ± 0.2
	Overall	7344	4.7
Medium	A	33% ± 8%	195.9 ± 4.3
	B	35% ± 10%	226.3 ± 4.8
	C	35% ± 8%	213.6 ± 4.7
	Overall	14385	212.6
High	A	29% ± 8%	609.7 ± 10.9
	B	35% ± 14%	727.9 ± 11.2
	C	34% ± 20%	662.1 ± 10.3
	Overall	21753	668.3

Table 6–5: Greedy Scheduling Under Uniform Performance

of length 12 and averaged over ten simulation runs, are shown in tables 6–5 and 6–6.

In the uniform case, greedy scheduling fails to find the optimal workload distribution of 33% per instance. The actual workload distributions observed varied considerably between simulation runs. This inconsistency is caused by a characteristic of greedy scheduling known as swamping.

Swamping occurs because the greedy scheduler always selects what it perceives to be the fastest instance, based on the average service time of recent invocations. When the arrival rate for the system exceeds the (observed) service rate of the fastest instance, more than one invocation may arrive between invocation completions, i.e., between status updates. Consequently, the status information becomes

Load Category	Object Instance	Invocations Received	Average Service Time
Low	A	1% ± 1%	7.0 ± 1.0
	B	4% ± 3%	3.1 ± 0.3
	C	95% ± 3%	1.0 ± 0.03
	Overall	7344	1.1
Medium	A	9% ± 4%	272.5 ± 9.8
	B	19% ± 5%	126.5 ± 3.7
	C	70% ± 6%	84.6 ± 1.3
	Overall	14385	109.1
High	A	19% ± 12%	1259.8 ± 20.4
	B	30% ± 11%	604.6 ± 10.4
	C	51% ± 15%	396.6 ± 5.4
	Overall	21753	592.0

Table 6–6: Greedy Scheduling Under Non-Uniform Performance

'stagnant' relative to invocation arrivals, with successive invocations being assigned to the same 'fastest' instance. The request queue of the fastest instance therefore fills up faster than the instance can service it. The observed service times gradually become worse, reflecting the time spent by each invocation in the request queue. Eventually, the *average* service time will increase until one of the other instances appears faster. At this point the faster instance is selected and is in turn swamped.

The effect of swamping on the service times is clearly seen in Table 6–5. Once the system-wide arrival rate exceeds the capacity of a single object, i.e., at medium load, the average service time shoots up dramatically. This is a result of invocation requests spending most of their time in large request queues awaiting service.

In the non-uniform case, at low load, nearly all invocations are assigned to the fastest instance, C. This returns reasonable performance since (by design) this load is within C's capacity. However, at medium load swamping starts to occur, although the resultant performance is still marginally better than that for random scheduling. At high load the greedy scheduler apportions the invocations very close to the optimal 16%, 34%, 50% split, but the swamping effect yields a performance considerably worse than that of random scheduling. In particular, at high load and with uniform performance, the average service time per invocation for greedy scheduling is over 100 times that for random scheduling. This is because the apparently optimal invocation split is the net result of swamping each instance in turn, rather than continually rotating between them.

In conclusion, greedy scheduling with a performance history of 12 invocations, performs worse than random scheduling under almost all circumstances, but particularly when the instances offer similar performance. The main reason for this is the swamping caused by always assigning to the perceived 'fastest' instance.

6.6 Comparison Scheduling

Comparison scheduling is an enhanced form of greedy scheduling that attempts to remove the swamping effect. The basic idea is to select the fastest instance only if it is 'significantly' faster than the next one, otherwise selecting at random.

In the general case there are assumed to be c instances (copies) of the same, idempotent service ranked in order by average (recent) performance. The fastest instance is denoted S_1, the second fastest S_2 and so on down to the slowest, S_c. Comparison scheduling compares S_1 with S_2. If S_1 is 'significantly' faster, then it is selected to receive the request (being 'significantly' faster than S_2 implies S_1 is also significantly faster than all the others). Otherwise, S_2 and S_3 are compared. If S_2 is 'significantly' faster, then one of S_1 or S_2 is selected uniformly at random. Otherwise, S_3 and S_4 are compared. The algorithm proceeds in this way until a significant comparison is found. Should none of the tests prove significant, then the algorithm automatically defaults to uniform random scheduling. In general, if S_i is 'significantly' faster than S_{i+1} (or if $i = c$, the number of instances), then one of the i instances $S_1 \ldots S_i$ is selected uniformly at random, i.e., each with probability $\frac{1}{i}$.

The main contribution of this chapter is to define rigorously the criterion *'significantly faster'*, as well as providing a mechanism for performing the comparisons. The formulation of the problem presented below assumes exponential service times. This assumption is not critical to the comparison scheduling mechanism. However, it does simplify the mathematics involved, enabling the use of standard statistical tables. Other distributions could be used in systems where the exponential distribution is not a good model of service times. The statistical background required for developing the exponential example is provided by the following section.

6.6.1 Statistical Background

Distribution Relationships

The following relationships between probability distributions are used in developing the exponential comparison scheduler.

Relation 1 : If

$$X_1, X_2, \ldots, X_n \sim exp(\mu)$$

then

$$\sum_{i=1}^{n} X_i \sim \Gamma(n, \mu)$$

i.e., if X_1 up to X_n follow an exponential distribution with rate μ, then their sum is distributed according to a Gamma distribution, with parameters n and μ.

Relation 2 : If

$$Y \sim \Gamma(n, \mu)$$

then

$$2\mu Y \sim \chi^2_{2n}$$

i.e., if Y follows a Gamma distribution with parameters n and μ, then $2\mu Y$ follows a chi-squared distribution with $2n$ degrees of freedom.

Relation 3 : If

$$S \sim \chi^2_n \text{ and } T \sim \chi^2_m$$

then

$$\frac{S/n}{T/m} \sim F_{n,m}$$

i.e., if S and T follow chi-squared distributions with degrees of freedom n and m respectively, then the ratio $\frac{S/n}{T/m}$ follows an F distribution with degrees of freedom n, m.

Hypothesis Testing

A statistical hypothesis is an assertion or conjecture about the distribution of one or more random variables. To perform a hypothesis test, two contrasting hypotheses are formulated; the null hypothesis, H_0, and the alternative hypothesis, H_1. The **Null Hypothesis** is the main focus of attention. Generally this is a statement that a parameter has a specified value. Often the phrase 'there is no difference' is used in its interpretation, hence the name 'null' hypothesis. The **Alternative Hypothesis** is a statement about the same parameter, specifying a different value or range of values from those in the null hypothesis. Rejection of the null hypothesis implies acceptance of the alternative hypothesis.

A hypothesis test examines the outcome of a statistical experiment for consistency with the null hypothesis, yielding a statement of the form : 'Assuming H_0 to be true, then the observed outcome has probability P'. Should P be very small, i.e., the observed outcome is very *un*likely given the assumption that H_0 holds, then this provides evidence that H_0 is in fact false, and that H_1 should be accepted instead. The smaller the value of P, the stronger the evidence to reject H_0. Typical values of P for rejecting H_0 are 0.1, 0.05, 0.025 and 0.01. If the value of P lies in this range then the test is said to be **Significant** at the 10% level, 5% level, 2.5% level or 1% level respectively, with 1% significance providing the strongest evidence for rejection. Larger values of P, i.e., greater than 0.1, imply that H_0 should not be rejected. Note that this is not evidence *for* H_0, it is simply lack of evidence *against* H_0.

In comparison scheduling, the null hypothesis is that the average service times of the two instances being compared are identical. The observed service times are examined for consistency with this assumption. The alternative hypothesis is that the average service times are (significantly) different. The precise formulation of H_0 and H_1, along with the testing mechanism for generating the value of P, are described below.

6.6.2 A Statistical Model of Invocation

The model of invocation used for comparison scheduling is the same as that described earlier for Random and Greedy scheduling (Figure 6–1). However, in order to simplify the mathematics involved, each object instance, S, is modelled as having a single service rate μ_s. This is an approximation to the model, since each instance is in fact composed of r_s invocation routines, each with its own (different) service rate $\mu_i (1 \leq i \leq r_s)$. As will be demonstrated later, this approximation does not impede invocation scheduling performance, because when comparing instances of the same object the 'error' is the similar for each of them.

The following discussion illustrates the comparison mechanism for two object instances A and B, which are assumed to have exponentially distributed service times. Instance A has service rate μ_a (unknown), and a history of service times is available for the previous n invocations, denoted $X_1, X_2, \ldots, X_n$. Similarly, instance B has service rate μ_b (unknown), and a history of service times for the previous m invocations, denoted $Y_1, Y_2, \ldots, Y_m$. In general, $n \neq m$.

By relation 1 :

$$\sum_{i=1}^{n} X_i \sim \Gamma(n, \mu_a)$$

$$\sum_{j=1}^{m} Y_j \sim \Gamma(m, \mu_b)$$

which, using relation 2, gives :

$$2\mu_a \sum_{i=1}^{n} X_i \sim \chi^2_{2n}$$

$$2\mu_b \sum_{j=1}^{m} Y_j \sim \chi^2_{2m}$$

and hence, applying relation 3 :

$$\frac{(2\mu_a \sum_{i=1}^{n} X_i)/2n}{(2\mu_b \sum_{j=1}^{m} Y_j)/2m} \sim F_{2n,2m}$$

This latter statement can be rearranged to give

$$\frac{\mu_a \bar{X}}{\mu_b \bar{Y}} \sim F_{2n,2m} \tag{6.1}$$

where $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ is the observed mean service time for instance A, and similarly, $\bar{Y} = \frac{1}{n}\sum_{i=1}^{m} Y_i$ is the observed mean service time for instance B.

The quantity $\frac{\mu_a \bar{X}}{\mu_a \bar{Y}}$ is known as the **Test Statistic**, and will be used to perform the hypothesis test. The probability of observing a particular value of the test statistic, i.e., P, can be found using standard statistical tables describing the F-distribution. However, whilst the values $\bar{X}$ and $\bar{Y}$ can be calculated from the observed service times, the values of μ_a and μ_b are unknown. This problem is solved by the formulation of H_0, which states that A and B offer identical performance, i.e.,

$$H_0 : \mu_a = \mu_b$$

against the alternative hypothesis that the service rates are different.

$$H_1 : \mu_a \neq \mu_b$$

Hence, under H_0, (6.1) becomes :

$$\frac{\bar{X}}{\bar{Y}} \sim F_{2n,2m} \tag{6.2}$$

6.6.3 The Comparison Scheduler

Under the null hypothesis of identical performance, (6.2) states that the ratio of the average service times of two object instances follows an F-distribution. This model can now be used as a basis for constructing a comparison scheduling algorithm.

Given several instances of an idempotent object, the scheduling problem is to decide which one of these instances should receive a newly generated invocation

request. In order to solve this problem the comparison scheduler keeps a history of service times for the previous h invocations per instance, in the same manner as the greedy scheduler. During the first few invocations, an instance's history will contain less than h observations, but this is accounted for by the degrees of freedom when performing the test (determining the value of h will be examined shortly). Using these performance histories, the comparison scheduler can calculate the current average service time for each instance, which are then ranked in order, fastest first. If not all instances have been invoked at least once, i.e., if the scheduler has *no* performance information for one or more instances, then random scheduling is used (alternatively, systematically use each instance once). The following description assumes that at least one observed invocation service time is available per instance.

The ratio of the average service times for the first and second fastest instances is calculated. The probability P of observing the actual value calculated, assuming H_0 is true, can be found using standard tables of the F-distribution. The degrees of freedom used to index these tables are $2n$ and $2m$, where n is the length of performance history for the fastest object, and m is the length of performance history for the second object ($n, m \leq h$). If P is small, i.e., under H_0 the observed ratio is very unlikely, then the null hypothesis is rejected. As indicated earlier, 'small' usually means $P < 0.1$. For example, if $0.1 > P \geq 0.05$, then the test is significant at the 10% level. If $0.05 > P \geq 0.025$ then the test is significant at the 5% level, and so on. Under these circumstances, the first object is deemed significantly faster than the second and is therefore selected to receive the invocation. Should the test *not* be significant, i.e., $P \geq 0.1$, then the first and second fastest instances are deemed to provide equal performance, and the algorithm moves on to compare the second and third instances.

The scheduler moves down the sorted list of average service times, comparing 2 with 3, 3 with 4 etc., until one of the tests is significant. At this point the

object instances can be partitioned into two groups; the 'fast' group — where all instances are deemed to provide equal performance — which is significantly faster than all members of the 'slow' group. One of the 'fast' group is then selected uniformly at random to receive the invocation. Should *none* of the tests prove significant, then all instances are deemed to provide identical performance and hence, the algorithm defaults to uniform random scheduling.

As an example, consider an invocation request for which there are six possible instances to choose from. Performance histories are available for all six instances, from which the average service times have been calculated and the instances ranked accordingly. In this example, assume that the first significant test is that between the third and fourth instances in the ranked list. The algorithm in Figure 6–4 proceeds as follows : A check is made to ensure that performance information is available on all contending instances. If not, then comparison testing cannot proceed, so random scheduling is used instead. In this example sufficient information is assumed available, so the comparisons begin. The average service time and history length for the fastest instance are determined. Upon entering the while loop, the corresponding figures are determined for the second fastest instance. The test statistic is then calculated by taking the ratio of the averages, which is then compared to the appropriate F-tables. In this case the test is not significant so the variables are reset for the next iteration, which compares the second and third fastest instances. Again the test is not significant so the loop is entered once more to compare the third and fourth fastest instances. This test *is* significant and so the loop terminates with `SplitPoint` set to 3. The remainder of the algorithm simply selects from the first, second and third fastest instances uniformly at random. Had no significant tests occurred, then the invocation would have been scheduled randomly between all six instances.

```
ComparisonSchedule(ThisInvocation : InvocationRequest);

     BEGIN
     IF { Not all instances invoked at least once }
          THEN RandomSchedule(ThisInvocation)
     ELSE BEGIN
          TopAverage := { Average service time of fastest instance };
          TopHistory := { No. observations used to calculate average };

          SplitPoint := 1
          SignificantlyBetter := FALSE;

          WHILE (NOT SignificantlyBetter) and { haven't compared all }
               DO BEGIN
               /* search for a significant test */
               NextAverage := { Average service time of next fastest };
               NextHistory := { No. observations used to calculate average };

               TestStatistic := NextAverage/TopAverage

               SignificantlyBetter := FSignificant(TestStatistic,
                                                   2*NextHistory,
                                                   2*TopHistory );

               IF (NOT SignificantlyBetter)
                    THEN BEGIN
                    /* set up for next iteration of loop */
                    TopAverage := NextAverage;
                    TopHistory := NextHistory;
                    SplitPoint := SplitPoint + 1;

                    { Move pointer on to next instance in sorted list }

                    END of IF;
               END of WHILE;

          IF SignificantlyBetter
               THEN BEGIN
               ChosenInstance := RandomBetween(1, SplitPoint);
               Invoke(ChosenInstance, ThisInvocation);

               END
          ELSE RandomSchedule(ThisInvocation);

          END of first ELSE;
     END of ComparisonSchedule;
```

Figure 6–4: The Comparison Scheduling Algorithm

6.6.4 Scheduling Parameters

Having described the comparison scheduling mechanism, it now only remains to define the significance level and history length to be used. Choosing the optimal significance level will be based on the simulation results described below. Several different levels — those normally used in hypothesis testing — are tested; specifically 10%, 5%, 2.5%, 1%, 0.5% and 0.1%. Significance levels outside this range are not normally used, with a 0.1% test being considered *very* highly significant. A suitable value for the history length is suggested by the F-distribution itself.

An F statistic has two degrees of freedom; ν_1 and ν_2. Therefore, finding the significant F-value for any given significance level involves looking up a table of values indexed by ν_1 and ν_2; a different table being used for each significance level. If the test statistic is *greater* than the tabulated value, then the test is significant. Beyond approximately 10 degrees of freedom the F-values become very similar. Consequently, standard F-distribution tables do not normally tabulate all possible combinations of ν_1 and ν_2. Interpolation can be used, where necessary, to find untabulated values. The tables used by the simulation program tabulate F-values for ν_1 and ν_2 in the range 1-10, 12, 24 and ∞. Although there is obviously a large range of values missing, these are not really required, since the F-distribution percentage points (F-values) for a test statistic with 24 degrees of freedom, are very similar to those for a test statistic with infinite degrees of freedom. A simple interpretation of this is that adding more degrees of freedom beyond 24 lends little or no additional accuracy to the hypothesis test.

In comparison scheduling, ν_1 and ν_2 correspond to twice the number of observations used when calculating the means of instances 1 and 2 respectively. The argument presented above suggests that a history of 12 observations per object provides sufficient information on which to base the comparisons. Consequently, the comparison scheduler only keeps information on the twelve previous invocations per object.

6.6.5 Simulation Results

The simulation configuration used is identical to that described earlier for random and greedy scheduling. There are three object instances, each with five invocation routines and service rates as defined earlier. The same workload files are used and, as before, two different cases are simulated; uniform and non-uniform performance. The service history retained for each instance is restricted to its twelve most recent invocations (or less if the instance has not yet completed twelve invocations).

All six combinations of low, medium and high workload with uniform and non-uniform instances are repeated for each of the significance levels 10%, 5%, 2.5%, 1%, 0.5% and 0.1%. The initial simulation results (only one simulation run each) are shown in Tables 6–7 and 6–8. Of the significance levels tested, 0.1% provides the best performance in all cases. The following analysis therefore focuses on this particular value.

The detailed simulation results for comparison scheduling with 0.1% significance, are shown in Tables 6–9 and 6–10. In all cases the average service times are considerably smaller than those for greedy scheduling. In particular, at medium and high loads, comparison scheduling yields performance an order of magnitude faster. Against the random scheduler's results, in the uniform case, comparison scheduling performs at least as well, whilst, in the non-uniform case, there is again an order of magnitude improvement. In all cases, the invocations are apportioned exactly according to each instance's servicing capabilities. There is no evidence in these service times of the swamping effect that afflicted greedy scheduling.

The column headed 'Actively Scheduled' indicates, for the invocations received, how many were the result of genuine selection as opposed to the default of random selection. In the uniform case (Table 6–9) at low load, all instances exhibit very similar performance, so the number of genuine selections is very low; approximately 1 in 10. Hence, under these circumstances comparison scheduling has

Load Category	Average Service Time at Significance Level :					
	10%	5%	2.5%	1%	0.5%	0.1%
Low	1.6	1.2	0.9	0.8	0.8	0.8
Medium	8.0	5.9	3.7	2.6	2.2	1.6
High	130.6	22.7	11.7	8.9	8.0	5.4

Table 6–7: Comparison Scheduling Under Uniform Performance

Load Category	Average Service Time at Significance Level :					
	10%	5%	2.5%	1%	0.5%	0.1%
Low	0.9	1.0	0.9	0.9	0.9	0.9
Medium	8.1	4.9	4.1	2.2	2.3	1.9
High	133.7	50.1	17.5	16.0	8.5	7.5

Table 6–8: Comparison Scheduling Under Non-Uniform Performance

Load Category	Object Instance	Invocations Received	Actively Scheduled	Average Service Time
Low	A	33% ± 2%	11% ± 5%	0.8 ± 0.04
	B	33% ± 2%	11% ± 5%	0.8 ± 0.04
	C	33% ± 2%	12% ± 5%	0.8 ± 0.04
	Overall	7344		0.8
Medium	A	33% ± 1%	26% ± 5%	1.6 ± 0.05
	B	33% ± 1%	25% ± 4%	1.7 ± 0.06
	C	33% ± 1%	26% ± 4%	1.7 ± 0.06
	Overall	14385		1.7
High	A	33% ± 1%	31% ± 3%	5.9 ± 0.1
	B	33% ± 1%	31% ± 1%	5.7 ± 0.1
	C	33% ± 1%	31% ± 4%	5.8 ± 0.1
	Overall	21753		5.8

Table 6–9: Comparison Scheduling (at 0.1%) Under Uniform Performance

Load Category	Object Instance	Invocations Received	Actively Scheduled	Average Service Time
Low	A	14% ± 5%	2% ± 2%	2.2 ± 0.2
	B	35% ± 2%	61% ± 17%	0.9 ± 0.04
	C	50% ± 4%	72% ± 14%	0.6 ± 0.02
	Overall	7344		0.9
Medium	A	15% ± 2%	3% ± 3%	4.8 ± 0.2
	B	35% ± 1%	57% ± 6%	1.9 ± 0.06
	C	50% ± 2%	70% ± 5%	1.3 ± 0.03
	Overall	14385		2.0
High	A	16% ± 1%	9% ± 5%	16.8 ± 0.4
	B	34% ± 1%	57% ± 5%	6.7 ± 0.1
	C	50% ± 1%	70% ± 3%	3.9 ± 0.06
	Overall	21753		6.9

Table 6–10: Comparison Scheduling (at 0.1%) Under Non-Uniform Performance

defaulted to uniform random behaviour. As the load increases and the request queues build up, the observed performances start to differ. This is reflected by an increase in the number of actively scheduled invocations to approximately 1 in 3 at high load. In the non-uniform case (Table 6–10), even at high loads the slow instance, A, is very rarely selected. Approximately 90% of the invocations it receives are the result of uniform random scheduling, i.e., when the performance of B and C has degraded to the extent that A now offers a comparable service. In contrast, the fastest instance, C, is actively selected for approximately 70% of all invocations it receives.

In conclusion, comparison scheduling, based on statistical hypothesis testing, adapts extremely well to the relative performance capabilities of multiple (three) object instances. In the uniform case, invocations are apportioned evenly between instances, emulating the actions of random scheduling. In the non-uniform case, comparison scheduling again apportions invocations exactly according to each instances' servicing capabilities. Initial experimentation with five object instances suggests these effects *improve* as the scale increases (or more accurately, random and greedy scheduling deteriorate, while comparison scheduling maintains performance). The excellent performance results obtained from these simple simulations confirm that the basic idea of using hypothesis testing merits further investigation.

6.7 Object Scheduling

The performance resulting from comparison scheduling of invocation requests, makes it worthwhile investigating the possibility of applying the same techniques to object scheduling, i.e., the assignment of object instances to object hosts. A model similar to that for invocation can be used, with objects replaced by hosts and invocation routines replaced by objects. Scheduling a new object instance then requires comparing the 'performance' of each host, using a hypothesis test

in the same manner as described for objects. A host is only selected if its average performance is significantly better than that of its peers, otherwise random selection is used.

One problem with this approach is in deciding how to measure host performance. With invocation scheduling, the average service times are compared for multiple instances of the *same* object, i.e., it compares 'like-with-like'. Calculating an average service time across all objects on the host does not yield this property. For example, one host may contain 'simple' objects, whose service times are inherently shorter than the services on a neighbouring host. This could lead to a bias towards the host with simple objects, even though its true performance may be no better than its peers. Further simulation work is required to establish whether anomalies such as this would have a detrimental effect upon object scheduling performance.

Object scheduling is further complicated by the need to consider construction costs. An executable representation may not exist for the 'fastest' host. Under these circumstances, rules must be applied to balance the trade-off between improved performance and cost of construction. For example, it may only be sensible to instigate construction of a new representation if the target host is significantly faster than *all* other hosts. Many other policies are possible. The detailed investigation of comparison scheduling of object instances is a topic for further investigation.

6.8 Summary and Conclusions

The novel approach of applying statistical hypothesis testing to scheduling, known as comparison scheduling, has been presented. For invocation scheduling, an instance is selected to receive an invocation only if it is significantly faster than

all other contenders. When no clear 'winner' can be found, then scheduling is performed at random. One particular formulation of this problem, based on exponential service times, was examined in detail. Many other formulations based on other distributions are possible. The exponential distribution was used primarily because it is the standard statistical model of service times, but also because mathematically it is easy to handle. The possibility of applying the comparison technique to object scheduling was also examined.

The simulation results clearly demonstrate the comparison mechanism's potential for use in scheduling algorithms. Performance improvements of up to an order of magnitude were observed when compared to the intuitive policies of random and greedy scheduling. The comparison scheduler adapted readily to changes in both workload and service capacity. For the exponential based comparison mechanism, the best performance was consistently achieved at a significance level of 0.1%.

The simplicity of the simulated environment limits the inferences that can be drawn from these results. However, they provide a strong indication that comparison scheduling offers a fruitful, new approach to scheduling in object oriented distributed systems.

Chapter 7

Status Updates

The comparison scheduler described in the previous chapter relies upon a rolling history of service times for the previous twelve invocations per object instance. As presented so far, it is assumed this history is updated after every invocation. In a large system with many objects, this represents a substantial overhead in both communication and processing costs. This chapter examines the problem of update suppression, i.e., reducing the number of status updates. Arbitrarily omitting, for example, every other message could lead to reduced scheduler performance caused by inaccurate or out-of-date information. What is required is a mechanism to suppress redundant messages, i.e., those offering little new information over and above that contained in previous updates. An algorithm for eliminating redundant updates is developed based upon the hypothesis testing techniques used in comparison scheduling.

7.1 Thresholding

The thresholding mechanism defined here is developed from the drafting algorithm of Ni *et al.* ([Ni 85]), which was described earlier in section 5.3.2 (page 132). The principle underlying the drafting algorithm is to generate an update only when there has been a notable change in load. Figure 7–1 shows a thresholding mechanism that emulates the drafting algorithm's behaviour. An update is generated when a processor's load crosses a boundary between two loading categories. However, when a processor's load lies close to a boundary it can oscillate between categories, generating continuous updates. This problem of 'state woggling' results from using fixed boundaries. The thresholding algorithm developed here maintains the principle of only generating an update when there is a notable change, but the threshold points are defined relative to the load reported in the previous update, rather than by predetermined, fixed categories.

Figure 7–2 demonstrates the general thresholding principle. It assumes an update was generated at time zero, reporting performance level P_{old}. At the same time as this update was generated, the two thresholds P_u and P_l were calculated as defined below. These thresholds divide the performance scale into three regions; a **Silent Region** and two **Update Regions**. The height of the silent region, known as the **Silence Interval**, is denoted by S. P_u and P_l are defined as functions of P_{old}, with perhaps the simplest example being $P_u = P_{old} + S/2$ and $P_l = P_{old} - S/2$. In general, there is no requirement that the silent region be symmetric about P_{old}.

No updates are generated while the current performance level lies within the silent region. However, when the performance reaches the level of one or other threshold (for example, at time t in Figure 7–2) an update is generated reporting the new current performance P_{new} (P_u in the example). Simultaneously, new thresholds P'_u and P'_l are calculated, for example, as $P_{new} + S/2$ and $P_{new} - S/2$

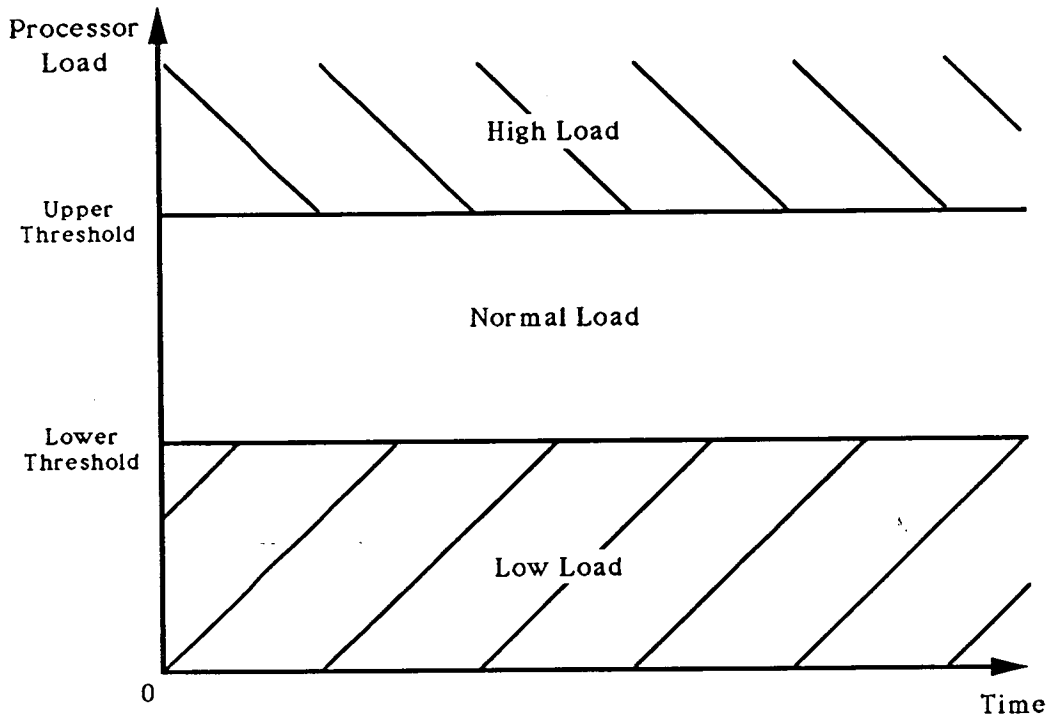

Figure 7–1: A Simple Thresholding Mechanism

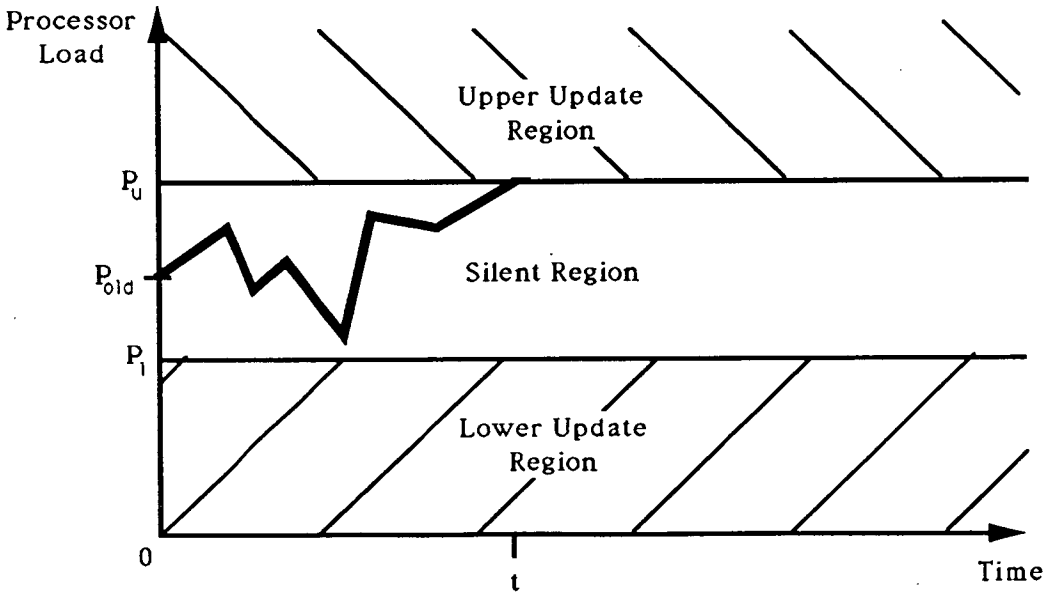

Figure 7–2: Thresholding Regions

respectively. The system then continues as before, using the new thresholds. Hence, an update is generated every time a threshold is crossed, and the thresholds themselves are re-calculated after every update. In this manner it is hoped to avoid the 'woggling' effect caused by using fixed thresholds.

Adjusting the height of the silent region, S, directly affects the rate at which updates are generated, thereby determining the accuracy, or granularity of the information they contain. A small value of S leads to a high update rate, while a large value of S generates fewer updates. In effect, S defines the term 'notable change', by placing a ceiling on the maximum change in performance that can go unreported.

7.2 Object Thresholding

The thresholding mechanism described above is completely general and could be applied not just to performance, but to any parameter that varies over time. In comparison scheduling, the parameter of interest is the average service time of an object instance. The following paragraphs define the silence interval S using the statistical hypothesis testing technique developed for comparison scheduling. The model of invocation assumed is identical to that used in the previous chapter. The example developed here therefore assumes exponentially distributed service times. However, as with comparison scheduling, this assumption is not fundamental to the technique; it simply serves to demonstrate the underlying mechanism, building upon the statistical model developed in chapter 6. The distribution relationships defined in that chapter will be used again here without being re-stated.

$X_1, X_2, \ldots, X_n$ represent the previous n service times observed for object instance A. The average service rate at the time of the previous update is denoted by μ_p. This value is estimated as the reciprocal of the average service time re-

ported in the update. The instance's current service rate is denoted by μ_c. By relation 1 :

$$\sum_{i=1}^{n} X_i \sim \Gamma(n, \mu_c)$$

which, using relation 2, gives :

$$2\mu_c \sum_{i=1}^{n} X_i \sim \chi^2_{2n} \tag{7.1}$$

The null hypothesis H_0 is that the current service rate is the same as the service rate at the time of of the previous update, i.e., no change in performance has occurred :

$$H_0 : \mu_c = \mu_p$$

against the alternative hypothesis that the rates are now different :

$$H_1 : \mu_c \neq \mu_p$$

Under H_0, (7.1) becomes :

$$2\mu_p \sum_{i=1}^{n} X_i \sim \chi^2_{2n}$$

This latter statement can be rearranged as :

$$2n\mu_p \bar{X} \sim \chi^2_{2n} \tag{7.2}$$

Under the null hypothesis of no change in performance, (7.2) states that the test statistic $2n\mu_p\bar{X}$ follows a chi-squared distribution with $2n$ degrees of freedom, where μ_p is the estimated service rate at the time of the previous update and $\bar{X}$ is the *current* average service *time* calculated over the previous n invocations. As with the comparison scheduling example, standard statistical tables, in this case chi-squared tables, can be consulted to yield a probability P for any particular observed value of the test statistic. A significant test (at some specified significance level) indicates that μ_p and μ_c are significantly different and that a new update should be generated. At this point the current value of $\bar{X}$ and n are passed to all

interested parties — principally the comparison scheduler — while the value of μ_p is re-estimated as $1/\bar{X}$. Should the hypothesis test not be significant, then no action is taken.

An alternative formulation of the same test, fitting more closely with the original description of thresholding, is to calculate a **Confidence Interval** around μ_p. A confidence interval is a range of values around a parameter estimate[1] indicating the 'accuracy' of the estimate. For example, a 95% confidence interval about μ_p defines a range of values within which the true service rate is predicted to lie with 95% certainty, i.e., with probability 0.95. Equivalently, the true value lies *outside* the confidence interval with probability 0.05 (Figure 7–3). Defining the threshold values P_u and P_l to be the bounding values of the confidence interval makes the statement 'the test is significant at the 5% level' synonymous with the statement 'the current performance lies outside the 95% confidence interval for μ_p', which in turn can be interpreted as 'the current performance level has entered an update region'. Hence, the confidence level of the confidence interval defines the height, S, of the silent region; a higher level such as 99%, specifying a larger silent region than a lower level such as 90%.

The update algorithm shown in Figure 7–4 uses the confidence interval approach. However, rather than calculating a confidence interval around μ_p, which follows a (relatively) complicated Gamma distribution, the test statistic $2n\mu_p\bar{X}$ is used. As shown in (7.2), this statistic follows a (simple) chi-squared distribution on $2n$ degrees of freedom, where n is the number of observations used to calculate the current average. The value of n becomes constant once the number of observations has reached the history length. Hence, for a 'running' system, i.e., one with more than n observations per object, the confidence interval for a given confidence level is constant. Under these circumstances, the term $2n\mu_p$, which is

---

[1] $\mu_p = 1/\bar{X}$ is only an estimate of the true, unknown, service rate

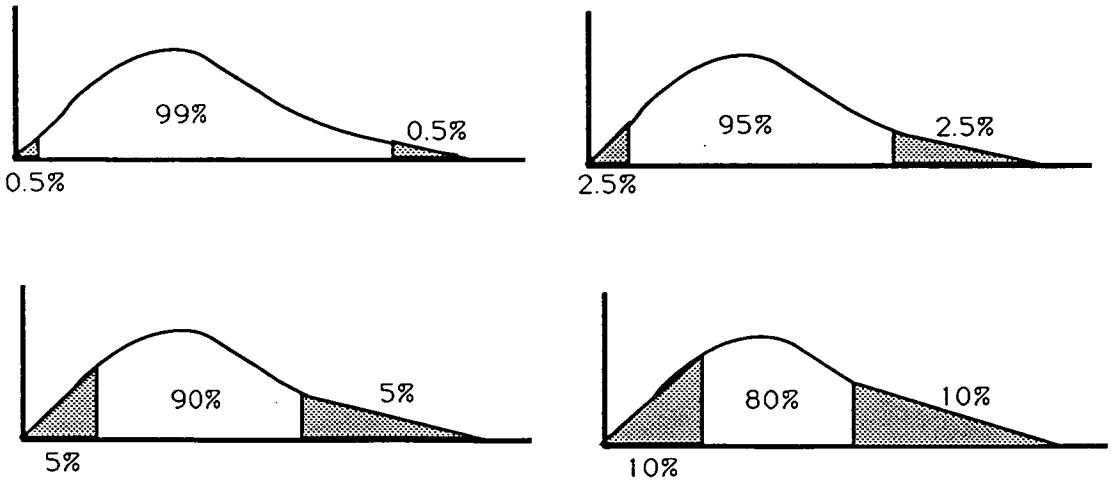

Figure 7–3: Confidence Intervals and Hypothesis Tests

also constant (between updates), can be thought of as a 'scaling factor', mapping the current performance level onto the range of values covered by the chi-squared distribution.

A range of confidence intervals about $2n\mu_p\bar{X}$ are tested, namely 99%, 95%, 90%, 80%, 60%, 40% and 20%. Table 7–5 shows the upper and lower bounds for these intervals based on 24 degrees of freedom, i.e, using a service history of size twelve as required by the comparison scheduler. The 99% level defines a large confidence interval (silent region), and consequently a low level of updates. Conversely, the 20% level defines a smaller confidence interval which should produce a higher update rate.

7.3 Simulation Description

The simulator used to test this update mechanism is the same as that used to test comparison scheduling. The only additions are performance monitors (Figure 7–6), one per instance, that collect the service time data and perform the thresholding test described above. Update messages are passed from the monitors to the sched-

```
UpdateSuppress(CurrentAverage : Real;
               HistoryLength  : Integer);

     BEGIN
     TestStatistic := 2*HistoryLength*PreviousUpdate*CurrentAverage;

     { calculate chi-squared confidence interval.  Depends only on}
     { the degrees of freedom, i.e., the number of observations   }
     { used to calculate CurrentAverage.                          }

     SetConfidenceInterval(LowerBound, UpperBound, 2*HistoryLength);

     IF (TestStatistic < LowerBound) OR (TestStatistic > UpperBound)
          THEN BEGIN
          SendUpdate(CurrentAverage, HistoryLength);
          PreviousUpdate := 1/CurrentAverage;

          END of IF;
     END of UpdateSuppress;
```

Figure 7–4: The Update Suppression Algorithm

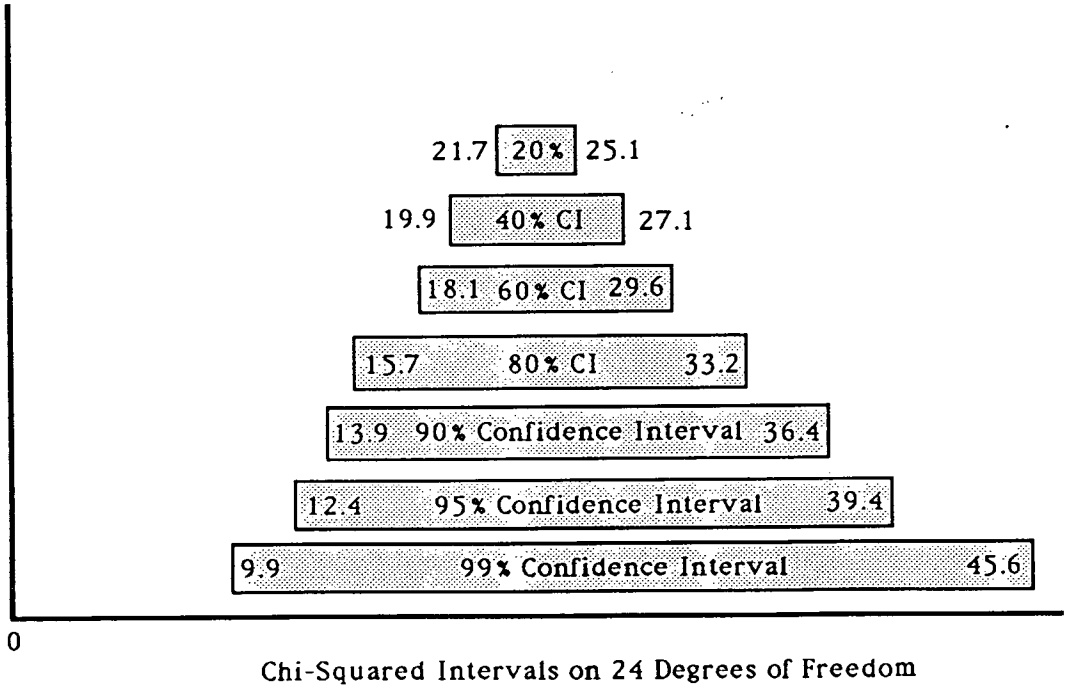

Figure 7–5: The Experimental Confidence Intervals

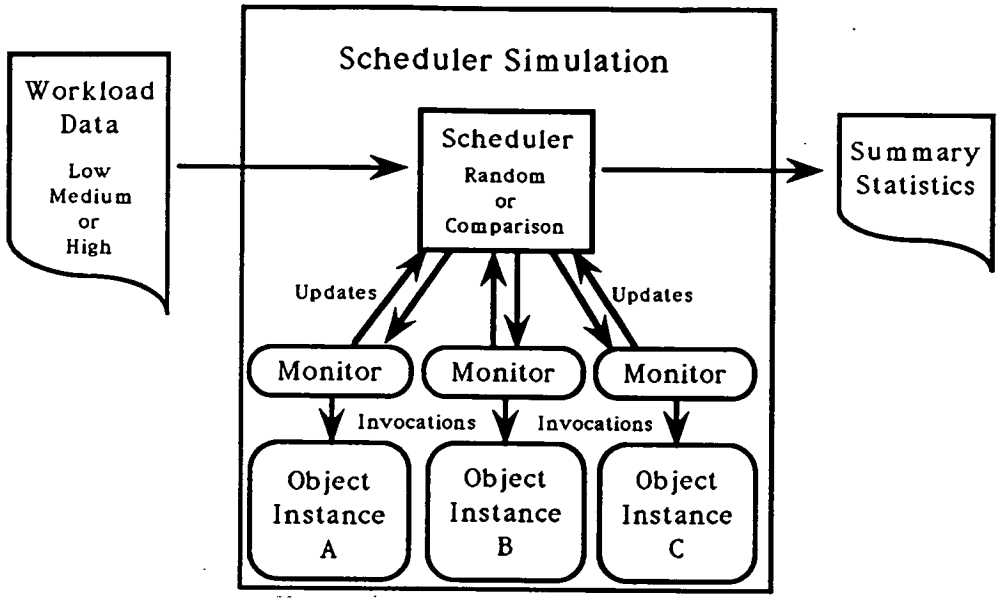

Figure 7–6: Simulator Configuration

uler, which now only keeps service averages and their associated history lengths, rather than individual observations. All other aspects of the simulation are the same as described in chapter 6.

7.3.1 Controlling Suppression

The initial simulation experiments investigated the relationship between the significance level of the test and the level of update suppression. The random scheduler was employed in order to eliminate any (unpredictable) mutual dependencies or feedback between the update level and the scheduler's performance. The results of these experiments are shown in Tables 7–1 and 7–2. An update rate of 100% implies an update is generated on every invocation, which was the default assumed in the previous chapter. A rate of 50% implies only one update (on average) for every two invocations, a rate of 25% implies one update (on average) for every 4 invocations and so forth.

Load Category	Object Instance	Updates with Interval Size :						
		99%	95%	90%	80%	60%	40%	20%
Low	A	5%	8%	10%	13%	21%	31%	53%
	B	5%	7%	10%	13%	22%	33%	52%
	C	5%	7%	10%	12%	21%	34%	53%
Medium	A	6%	9%	10%	15%	23%	35%	59%
	B	5%	8%	11%	15%	23%	36%	59%
	C	6%	9%	11%	14%	24%	34%	58%
High	A	3%	5%	5%	7%	12%	23%	32%
	B	3%	5%	6%	7%	15%	22%	38%
	C	3%	4%	7%	8%	13%	10%	32%

Table 7–1: Update Suppression with Uniform Performance

Load Category	Object Instance	Updates with Interval Size :						
		99%	95%	90%	80%	60%	40%	20%
Low	A	5%	9%	12%	14%	23%	36%	58%
	B	4%	7%	9%	12%	21%	33%	50%
	C	4%	7%	8%	12%	18%	30%	50%
Medium	A	0%	0%	0%	0%	1%	2%	3%
	B	6%	8%	11%	15%	21%	35%	59%
	C	5%	8%	10%	14%	22%	34%	55%
High	A	0%	0%	0%	0%	0%	1%	2%
	B	3%	6%	5%	8%	13%	19%	37%
	C	6%	8%	11%	14%	22%	35%	58%

Table 7–2: Update Suppression with Non-Uniform Performance

These figures confirm that the suppression mechanism performs as expected. For any given instance there is a linear reduction in the number of updates generated as the confidence interval increases. There is also a relationship between the update rate and the workload, or more accurately, between the update rate and the average service time. This is particularly in evidence in the non-uniform case, where the slowest instance, A, produces very few updates at medium and high loads. This is a consequence of the very high average service times[2] resulting from the random scheduler continually overloading A. The 'scaling factor' $2n\mu_p$, where μ_p is the reciprocal of the previously reported (poor) performance, means that large average service times require a proportionally larger change in performance before a significant test is encountered. Hence the reduced number of updates for A, and likewise, the increase in updates for C (B remains the same as in the uniform case).

7.3.2 Scheduler Performance

Having established a qualitative link between the update rate and the threshold significance level, the next series of simulations examine the effect of reduced information upon comparison scheduling performance. The simulation configuration is exactly as before, except that the random scheduler is replaced by the comparison scheduler with a comparison significance level of 0.1%, as determined in the previous chapter. The initial results (one run each) are given in Tables 7–3 to 7–8. The corresponding results for the full information cases, taken from Tables 6–9 and 6–10, have been included for comparison.

Concentrating on the maximum suppression case (99% confidence interval), the reduced update rate has very little impact upon the comparison scheduler's

[2]See Tables 6–1 and 6–2 for the service times associated with random scheduling.

Significance Level	Object Instance	Update Rate	Invocations Received	Average Response
99%	A	5%	32%	0.8
	B	5%	35%	0.8
	C	5%	33%	0.8
	Overall		7344	0.8
95%	A	8%	33%	0.8
	B	8%	33%	0.8
	C	8%	34%	0.7
	Overall		7344	0.8
90%	A	10%	33%	0.8
	B	10%	33%	0.8
	C	10%	34%	0.7
	Overall		7344	0.8
80%	A	13%	34%	0.7
	B	14%	33%	0.8
	C	14%	33%	0.8
	Overall		7344	0.8
60%	A	21%	33%	0.8
	B	21%	33%	0.7
	C	22%	33%	0.8
	Overall		7344	0.8
40%	A	35%	33%	0.9
	B	35%	33%	0.8
	C	34%	33%	0.8
	Overall		7344	0.8
20%	A	53%	34%	0.7
	B	53%	33%	0.8
	C	55%	33%	0.8
	Overall		7344	0.8
Full Information Case	A	100%	33%	0.8
	B	100%	33%	0.8
	C	100%	33%	0.8
	Overall		7344	0.8

Table 7–3: Update Suppression with Uniform Performance at Low Load

Significance Level	Object Instance	Update Rate	Invocations Received	Average Response
99%	A	6%	33%	1.7
	B	6%	34%	1.6
	C	6%	33%	1.6
	Overall		14385	1.6
95%	A	10%	33%	1.9
	B	10%	34%	1.7
	C	10%	34%	1.7
	Overall		14385	1.8
90%	A	12%	34%	1.8
	B	11%	32%	1.8
	C	13%	34%	1.8
	Overall		14385	1.8
80%	A	16%	34%	1.8
	B	16%	33%	1.9
	C	17%	33%	2.0
	Overall		14385	1.9
60%	A	25%	34%	1.6
	B	25%	33%	1.5
	C	24%	33%	1.6
	Overall		14385	1.6
40%	A	38%	33%	1.8
	B	39%	33%	1.7
	C	36%	34%	1.6
	Overall		14385	1.7
20%	A	60%	34%	1.6
	B	62%	33%	1.7
	C	61%	33%	1.6
	Overall		14385	1.6
Full Information Case	A	100%	33%	1.6
	B	100%	33%	1.7
	C	100%	33%	1.7
	Overall		14385	1.7

Table 7–4: Update Suppression with Uniform Performance at Medium Load

Significance Level	Object Instance	Update Rate	Invocations Received	Average Response
99%	A	5%	33%	6.8
	B	5%	34%	6.5
	C	5%	33%	7.4
	Overall		21753	6.9
95%	A	7%	34%	5.7
	B	7%	32%	6.2
	C	7%	34%	5.6
	Overall		21753	5.8
90%	A	8%	33%	6.7
	B	8%	33%	5.9
	C	9%	34%	5.7
	Overall		21753	6.1
80%	A	10%	33%	7.2
	B	11%	34%	7.2
	C	10%	33%	7.1
	Overall		21753	7.2
60%	A	18%	34%	6.0
	B	18%	33%	6.4
	C	17%	33%	6.7
	Overall		21753	6.4
40%	A	26%	34%	7.2
	B	25%	33%	7.4
	C	26%	34%	6.9
	Overall		21753	7.1
20%	A	49%	33%	5.3
	B	49%	34%	5.7
	C	50%	33%	5.8
	Overall		21753	5.6
Full Information Case	A	100%	33%	5.9
	B	100%	33%	5.7
	C	100%	33%	5.8
	Overall		21753	5.8

Table 7–5: Update Suppression with Uniform Performance at High Load

Significance Level	Object Instance	Update Rate	Invocations Received	Average Service Time
99%	A	6%	13%	2.5
	B	6%	37%	0.9
	C	5%	50%	0.6
	Overall		7344	0.9
95%	A	8%	13%	2.3
	B	9%	36%	0.9
	C	8%	51%	0.6
	Overall		7344	0.9
90%	A	11%	10%	2.5
	B	10%	38%	0.9
	C	11%	52%	0.5
	Overall		7344	0.9
80%	A	15%	13%	2.3
	B	14%	36%	1.0
	C	13%	51%	0.5
	Overall		7344	0.9
60%	A	24%	15%	2.0
	B	22%	35%	1.0
	C	23%	50%	0.6
	Overall		7344	0.9
40%	A	37%	12%	2.4
	B	37%	35%	1.0
	C	36%	53%	0.6
	Overall		7344	1.0
20%	A	59%	18%	2.0
	B	56%	35%	0.9
	C	54%	47%	0.6
	Overall		7344	0.9
Full Information Case	A	100%	14%	2.2
	B	100%	35%	0.9
	C	100%	50%	0.6
	Overall		7344	0.9

Table 7–6: Update Suppression with Non-Uniform Performance at Low Load

Significance Level	Object Instance	Update Rate	Invocations Received	Average Service Time
99%	A	6%	10%	7.0
	B	7%	36%	2.3
	C	7%	53%	1.4
	Overall		14385	2.3
95%	A	10%	14%	5.1
	B	11%	34%	1.9
	C	11%	52%	1.3
	Overall		14385	2.0
90%	A	13%	14%	5.9
	B	13%	35%	2.3
	C	12%	51%	1.4
	Overall		14385	2.4
80%	A	17%	14%	5.1
	B	15%	34%	2.2
	C	17%	51%	1.4
	Overall		14385	2.2
60%	A	24%	15%	4.6
	B	26%	34%	2.2
	C	25%	51%	1.2
	Overall		14385	2.0
40%	A	39%	16%	4.5
	B	40%	35%	1.9
	C	39%	50%	1.3
	Overall		14385	2.0
20%	A	64%	11%	5.2
	B	61%	35%	2.0
	C	62%	54%	1.4
	Overall		14385	2.0
Full Information Case	A	100%	15%	4.8
	B	100%	35%	1.9
	C	100%	50%	1.3
	Overall		14385	2.0

Table 7–7: Update Suppression with Non-Uniform Performance at Medium Load

Significance Level	Object Instance	Update Rate	Invocations Received	Average Service Time
99%	A	3%	17%	23.9
	B	5%	34%	9.7
	C	5%	49%	6.0
	Overall		21753	10.3
95%	A	6%	16%	16.4
	B	7%	33%	6.8
	C	8%	50%	3.9
	Overall		21753	6.9
90%	A	8%	15%	19.2
	B	9%	34%	8.1
	C	8%	51%	4.9
	Overall		21753	8.2
80%	A	10%	15%	21.4
	B	10%	33%	8.8
	C	10%	52%	4.9
	Overall		21753	8.7
60%	A	15%	17%	16.7
	B	17%	33%	7.0
	C	18%	50%	4.0
	Overall		21753	7.2
40%	A	23%	16%	19.3
	B	26%	34%	7.5
	C	27%	50%	4.5
	Overall		21753	7.8
20%	A	52%	16%	14.5
	B	51%	34%	5.9
	C	51%	50%	3.5
	Overall		21753	6.1
Full Information Case	A	100%	16%	16.8
	B	100%	34%	6.7
	C	100%	50%	3.9
	Overall		21753	6.9

Table 7–8: Update Suppression with Non-Uniform Performance at High Load

performance. At low and medium workloads, despite a 95% reduction in update traffic, performance remains identical to that for the full information case. At high workload there is some degradation in performance; 6.9 as compared to 5.8 in the uniform case, and 10.3 as compared to 6.9 in the non-uniform case, but this must be traded against the reduction in update traffic of approximately 95% (almost 21000 messages in the simulated example), representing a considerable saving in resource costs.

Despite the reduction in performance at high loads caused by reduced information, comparison scheduling with update suppression still yields performance an order of magnitude faster than both random and greedy scheduling, (see Tables 6–2, 6–5 and 6–6, operating with full information). The detailed results, averaged over ten simulation runs, for comparison scheduling with a comparison significance level of 0.1%, and update suppression with a thresholding confidence interval of 99%, are shown in Tables 7–9 and 7–10. These results suggest that there is a considerable amount of redundant information obtained when observing the service times for *all* invocations, and that the thresholding mechanism developed in this chapter performs well at removing this redundancy.

7.4 Service Expansion and Contraction

Thresholding can also be used to control service expansion and contraction, as described in chapter 5. Upper and lower thresholds, defined by the confidence level, can be placed on 'acceptable' service times. Should a single instance become overloaded, so that its average service time exceeds the upper threshold, then the creation of an additional instance is triggered in order to handle some of the load. Conversely, when the workload subsequently falls, the multiple instances become under-utilised and their average service times fall. If the lower threshold is passed, then a 'garbage collector' can be triggered to remove redundant instances, with

Load Category	Object Instance	Update Rate	Invocations Received	Average Service Time
Low	A	5% ± 1%	33% ± 2%	0.8 ± 0.04
	B	5% ± 1%	33% ± 2%	0.8 ± 0.04
	C	5% ± 1%	33% ± 2%	0.8 ± 0.04
	Overall	7344		0.8
Medium	A	7% ± 1%	33% ± 1%	1.8 ± 0.06
	B	7% ± 1%	33% ± 2%	1.8 ± 0.06
	C	7% ± 1%	33% ± 1%	1.9 ± 0.06
	Overall	14385		1.8
High	A	5% ± 1%	33% ± 1%	7.4 ± 0.2
	B	5% ± 1%	33% ± 1%	7.0 ± 0.1
	C	5% ± 1%	33% ± 1%	6.8 ± 0.1
	Overall	21753		7.1

Table 7–9: Comparison Scheduling with Update Suppression (Uniform)

Load Category	Object Instance	Update Rate	Invocations Received	Average Service Time
Low	A	6% ± 1%	14% ± 3%	2.3 ± 0.2
	B	6% ± 1%	37% ± 3%	0.9 ± 0.04
	C	6% ± 1%	50% ± 3%	0.6 ± 0.02
	Overall	7344		1.0
Medium	A	7% ± 1%	14% ± 2%	5.7 ± 0.3
	B	7% ± 1%	35% ± 1%	2.2 ± 0.07
	C	7% ± 1%	51% ± 2%	1.4 ± 0.04
	Overall	14385		2.3
High	A	4% ± 1%	16% ± 1%	22.4 ± 0.6
	B	4% ± 1%	33% ± 1%	9.1 ± 0.2
	C	4% ± 1%	50% ± 1%	5.6 ± 0.09
	Overall	21753		9.4

Table 7–10: Comparison Scheduling with Update Suppression (Non-Uniform)

the future workload being shared among remaining instances. Using thresholding in this way controls the level of utilisation of object instances.

7.5 Implementation Considerations

This section considers how comparison scheduling and update thresholding might be implemented in a distributed environment.

Every object host is capable of supporting at least one object (by definition), and every object may potentially generate invocations upon other objects. In general, therefore, each object host must be capable of supporting both status updates, for reporting the performance of the objects it supports, and comparison scheduling, for scheduling the invocations that its objects generate. Conceptually, both of these services must be replicated on a per object basis. However, in practice, they are more likely to be engineered as one replicate per host, with each replicate supporting all of one host's objects. Scheduling and update reporting are unlikely to be engineered in a centralized manner.

Replicating the scheduling and update *algorithms* is relatively simple. They can be included as part of the support environment for a host. The problems arise when deciding how to replicate the performance *data* that drives the algorithm, i.e., how to direct status updates to those hosts that require them.

For object scheduling, where the status information relates to host performance, it may be possible to update all hosts, especially if the underlying communication facilities support inexpensive broadcasts. The limiting factors to this approach are the number of hosts present in the system, and the frequency with which their status is updated. With the significantly reduced update rate resulting from use of the update mechanism developed in this chapter, total monitoring may be feasible for the size of systems defined in chapter 3. However, for larger systems

it may be necessary to partition hosts into 'schedule groups', where members of the same schedule group monitor each other, but not members of other groups. This approach, namely reducing a large, unmanageable system to multiple, smaller, self-contained systems, is examined further in chapter 9 when considering the configuration management of Environments.

For invocation scheduling, the problem of distributing *object* performance data is potentially much greater, since there are assumed to be many more objects than hosts. However, for any particular invocation, the candidate objects are not those of the entire system, but rather, only those that support the specific service being invoked. Hence, for invocation scheduling, each host only actually needs to maintain status information relating to objects that support the services required by the objects it hosts. If it were possible to identify these supporting objects, then the amount of information that must be held by each host would be considerably reduced.

It may be possible for a host to determine the services an object may (potentially) invoke, by examining the object itself. For example, when a new object is constructed, this sort of information may be recorded by the transformation tools. Hence, the host can then join the collection of hosts receiving updates about these services. The host is then in a position to schedule any future invocations that may be generated. If, however, the supporting services required by a new object can *not* be determined in advance, then the host must register for performance updates dynamically, as each new service is invoked for the first time.

7.6 Conclusions

The simulation results suggest that update suppression using the thresholding mechanism yields a considerable reduction in update traffic, while having little or no detrimental effect upon the comparison scheduler's performance. Assuming exponentially distributed service times, a 0.1% comparison significance level, together with a thresholding confidence interval of 99%, provides excellent scheduling performance across all configurations tested, combined with a 95% reduction in update traffic when compared to the 'complete information case'.

Chapter 8

Virtual Objects

This chapter identifies several example virtual properties that can be created by suitable manipulation of service instances and the invocations upon them. Virtual templates are introduced as a general, re-usable mechanism for creating this type of virtual property. Mock template implementations are provided for Resilience, Persistence, Access Controls, Inter-Object Debugging and Performance Monitoring. The chapter concludes by examining some of the implementation issues associated with creating and applying templates.

8.1 Virtual Properties

As defined in chapter 2, virtual properties are characteristics possessed by objects, that somehow 'improve' the service offered, but in a service independent manner. Typically, virtual properties mask inherent limitations of the underlying system or enhance an object's interface to match that expected by its invokers. An object displaying no virtual properties is known as a **Base Object** or **Real Object**. An object enhanced by the addition of one or more virtual properties is known as a **Virtual Object**. This chapter examines a particular class of virtual properties that can be retrospectively added to an object, either by manipulating invocations upon instances of the object, or by manipulating the instances themselves; or possibly a combination of the two. The following sections identify several properties commonly found in distributed systems, that comply with this definition.

8.1.1 Resilience

A K-resilient object is one whose invocation routines are guaranteed to progress to completion, despite the 'failure' of up to K hosts in the system. The notion of host failure covers any error condition denying access to the host's services, for example, the failure of the host itself, or a network partition restricting remote access.

Creating a K-resilient virtual object using replication requires at least $K+1$ independent instances of the object to exist within the system, where independence refers to their failure modes [Birma85]. Two objects are considered independent if the failure of one does not automatically imply the failure of the other. In practice this relates to failures of object hosts. Independence is therefore usually achieved by assigning each instance to a separate processor. Having created $K+1$ independent instances, each instance maintains its internal state in step with

the others by forwarding details of all invocations it receives. The uniform view of the service's state, as maintained by each instance, means that any one of the multiple instances is capable of responding to invocation requests for their particular service.

In the event of a failure, the failed instance no longer provides a service. However, the surviving K instances continue to function as normal. Clients remain oblivious to any problems, perceiving only a continuous, uninterrupted service. Up to K successive failures can be tolerated in this manner before the service fails altogether. Hence, although the individual real objects are not resilient to failure, a service displaying the resilience property is abstracted from their combined behaviour.

The key to realising K-resilience is the appropriate coordination of invocations upon the $K+1$ base object instances. A service endowed with K-resilience is enhanced by its ability to continue service provision despite the presence of up to K failures in the system.

8.1.2 Persistence

A concise definition of persistence is provided by Low, who states that :

> An object is persistent if it 'dies at the right time'; persistence is an observation about the lifetime of an object, namely that it exists for precisely as long as intended, and then it disappears. A programmer demands that an object disappears as an act of intention, not as a result of a processor crash or a bad block on a disc drive [or any other form of 'accidental' death]. [Low 88]

In the event of a host failure, the internal state of a persistent object is 'suspended' until the host recovers. At the point of recovery, the object and its internal state

are returned to their condition prior to the failure. Resilience and persistence are both related to reliability issues since resilience can be viewed as 'short term reliability', and persistence as 'long term reliability'. A persistent service may become temporarily unavailable during a partial system failure, whereas a resilient service continues to function. However, persistent objects can survive a total failure of all hosts, whereas resilient objects do not.

Persistent objects are realised by retaining an independent record of the object's internal state on a reliable storage medium [Cocks84]. If a failure occurs, this record can be used to restore the object to its former state. In an object-oriented environment, storing a trace of all invocations made upon an object is sufficient to ensure it can be reinstated. Upon host recovery after a failure, a new object instance is created to which the invocation list is then replayed. Upon completion of this invocation replay, the new object's state should re-create that of its failed predecessor.

Persistence can therefore be created through monitoring invocations upon an object, and copying them to stable storage for subsequent replay in the event of a failure. An object thus endowed with persistence is enhanced by the longevity of its internal status.

8.1.3 Access Control

Computer systems normally require some form of security mechanism to govern access to services. In a system consisting entirely of objects, this principally involves the verification of access rights when objects are invoked. Two methods commonly used to implement this are access control lists (ACLs) and capabilities [Mizun87], which were described earlier in section 1.8 in relation to the Cronus and Amoeba distributed systems.

It is possible for access controls on invocations to be checked externally to the service object, using a security object interposed between the client and server. Clients invoke the security object rather than the base object, presenting the appropriate security information for validation. Approved requests are then forwarded by the security object to the base object. Invocations failing the security validation are rejected. This approach to security, i.e., retrofitting to an existing system, known as **Incremental Addition** [Karge88], assumes clients cannot defeat the access controls by circumventing the security object and calling the base object directly.

Subject to the restriction of non-circumvention, access control mechanisms can be added to existing object instances by manipulation of invocation requests; more specifically, by interposing a security object between the client and server. There are several advantages to this approach : the overhead of secure access is only borne by those services requiring it; by using different security objects, multiple instances of the same base object may operate with different access control mechanisms; and finally, attaching more than one security object to a base object allows it to be simultaneously invoked by clients using different security mechanisms. For example, a single instance can appear to support both ACL based access control *and* capability based access control.

8.1.4 Debugging

Debugging in a distributed environment is intrinsically more complicated than in a non-distributed, sequential environment. There are a number of reasons for this [Garci84], [Joyce87] :

- Multiple foci of control
- Determination of current (distributed) state

- ▷ Inherently non-deterministic
- ▷ Debugging alters behaviour
- ▷ Complex (parallel) interactions between components

Traditional single user, sequential debuggers offer facilities such as single stepping, breakpoints and source code manipulation. These techniques, which can be thought of as **Intra-Object Debugging** aids, still apply when debugging individual objects. However, in a distributed environment, further facilities are required to debug erroneous *interactions* between objects, i.e., **Inter-Object Debugging**. This requires monitoring inter-object communications, i.e., invocation messages.

In [Smith85], Smith describes a debugger operating on message based, communicating processes; a definition that encompasses the object oriented invocation paradigm. This debugger provides mechanisms for accessing and controlling the inter-process activities of the system. Messages can be intercepted, stored for replay, modified, or even destroyed. The debugger deliberately does not provide any facilities for examining or manipulating information at a finer grain, such as the code or data of individual object instances. The Amoeba system provides a debugger operating on similar principles [Elsho88], although in this case access is also provided to the internals of an object, requiring it to be re-compiled with a `debug` attribute.

The debugging of object interaction, i.e., inter-object debugging, can therefore be provided through monitoring and manipulating the invocations made upon an object instance. Understanding program behaviour may be improved by observing the interactions and their associated parameters; errors can be reproduced by replaying previously stored invocation histories; while modifying parameters and results allows the programmer to experiment with the object's behaviour. All these facilities can be provided without any explicit cooperation from the objects

being debugged, and without the need to create special 'debuggable' instances (although this could be done at the same time as a separate operation).

8.1.5 Performance Monitoring

The performance information used to drive the update thresholding mechanism described in the previous chapter, is derived solely from observing invocation arrival times and reply departure times. Based on this information, status updates are generated according to the thresholding principle. Performance monitoring or 'monitorability' as defined in chapter 7 therefore complies with the definition of a virtual property. An object enhanced with monitorability enables any interested party to keep a record of the object's performance history.

8.2 Virtual Mapping

In each of the examples described above, the virtual property is created by manipulating base object instances and the invocations made upon them. This manipulation can be performed by interposing a virtual property object between the client and server objects, both of which remain unaware of the property object's existence. The property objects perform a mapping function, mapping from an 'ideal' environment of virtual objects onto the underlying 'warts-and-all' real environment. The base objects do not participate in creating virtual properties other than by providing their normal service interface.

The Proxy Principle [Shapi86], described in section 2.7.5 (page 70), provides an example of a general mapping mechanism that could be used to implement virtual properties. Proxies provide a single entry point to a service which, in reality, may be constructed from multiple distributed objects. The complexity of managing and coordinating the objects is encapsulated within the proxy and is

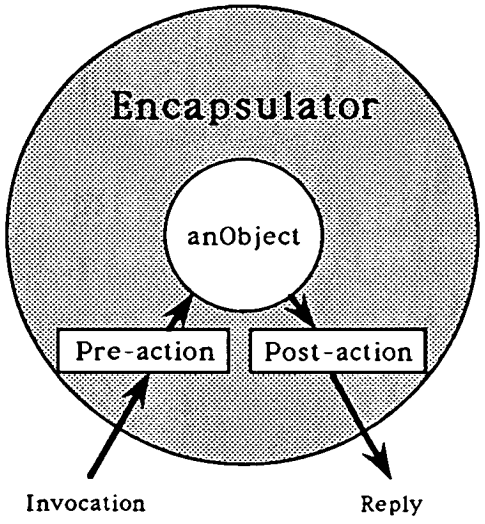

Figure 8–1: The Encapsulator Paradigm

thereby hidden from the service's clients. In general, a proxy is specific to the group it represents, i.e., each service has its own specially coded, unique proxy.

Encapsulators [Pasco86], implemented within a Smalltalk-80 environment, also embody some of these ideas. When an invocation is made upon an encapsulated object, the encapsulator performs a pre-action before the object is invoked and a post-action before a result is returned (Figure 8–1). By suitably defining the pre- and post-actions, a range of properties can be realised. Two examples given in [Pasco86] are mutual exclusion and atomic updates.

Realising mutual exclusion using an encapsulator is extremely simple. The pre-action performs a semaphore *wait*, while the post-action performs a semaphore *signal*. This ensures that only one invocation can proceed at a time, thus providing exclusive access to the encapsulated object on a per-invocation basis. For atomic updates, the pre-action creates a copy of the encapsulated object and passes the invocation to the copy. Upon successful completion of the invocation, the post-action uses Smalltalk's `become:` primitive to atomically replace the original object with the updated copy.

Pascoe suggests that the encapsulator paradigm has applications in distributed systems for implementing access to remote objects, or for implementing security mechanisms. However, since these applications have little relevance to the Smalltalk-80 single user, single machine environment they were not explored further.

8.3 Virtual Templates

Virtual templates combine the beneficial attributes of encapsulators (generality) and proxies (hiding distribution), to provide a general mechanism for mapping virtual objects to real objects. A **Virtual Template** takes the role of the virtual property object identified earlier — encapsulating, in a reusable form, the invocation and instance manipulations necessary to realise a particular property. The purpose of each virtual template is to provide a single virtual property applicable, in theory, to *any* base object. Thus, it is envisaged there will be a resilience template, a persistence template, a capability template, an access-control-list template, and so on. Any service requiring, for example, capability based access controls, can be created by applying the capability template to the appropriate base object. The resultant virtual object is identical to the base object except for the need to present a valid capability with each invocation. Similarly, a resilient version of the service can be created by applying the resilience template to the base object.

Using a paradigm such as virtual templates to encapsulate virtual properties in a reusable form, reduces the complexity associated with programming in a distributed environment. New objects can be coded without concern for issues such as reliability and access controls. When required, these properties can be added later by applying the appropriate virtual templates.

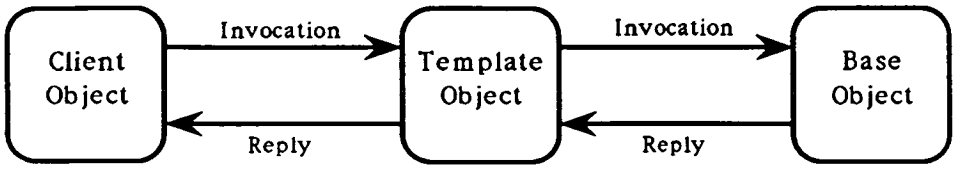

Figure 8–2: Interposing a Template Object between Client and Server

8.4 Template Operation

The following sections describe the template actions required to realise the example virtual properties identified earlier. The mock implementations presented are pedagogical; the algorithms shown are simplified, concentrating on readability rather than implementation details. For the same reason, the programming constructs used do not correspond to any particular language.

For the moment it is assumed that when a template is applied, the name of (i.e., a pointer to) the base object upon which the template is to operate is passed as a parameter. The base object instance is assumed to exist already. Invoking `Service Initialisation` on the template then establishes the virtual property, in some cases creating additional base object instances. The template is considered to be an independent object, invisibly interposed between the client and base object(s) (Figure 8–2). All invocations directed at the base object are assumed to be (automatically) redirected to the template. Section 8.5 provides further details on implementing and applying templates (as opposed to implementing the algorithms they execute).

8.4.1 Resilience

Creating a K-resilient service through replication requires at least $K + 1$ coordinated copies of the base object. In order to present a consistent service, details

of invocations upon any one copy must be forwarded, in the correct order, to the other K copies. The function of the resilience template is therefore principally to maintain consistency. In order to do this, the template requires support for the correct ordering of invocations upon replicates. Possible approaches include building upon atomic transactions or using ordered broadcasts. As an aside, the template also controls creation and deletion of the multiple base object instances to respectively create and delete the virtual service.

An example resilience template implementation is shown in Figure 8–3. The template adds five extra invocation routines to the interface presented by the base object : `Service Initialisation`, `Set Cohorts`, `Service Provision`, `Service Update` and `Service Closedown`. These additional routines are only used by the virtual property 'management' system (e.g., other resilience templates), and remain hidden from normal clients.

Service Initialisation

Procedure `Service_Initialisation` is called once in order to turn the original, non-resilient `Base_Object + Template` combination into a K resilient service. It does this by creating an additional K instances of the `Base_Object`, each with its own (identical) resilience template to cooperate in the coordination of service invocations. The algorithm assumes this routine is *not* called automatically, otherwise each of the new templates would in turn create K new instances, *ad infinitum*. `Find Independent Host()` is assumed to be a facility provided by the system's scheduling mechanism, returning the name of a host that is independent of those supporting the cohorts established so far. The statement

```
NEW Resilience_Template(NEW Base_Object) @ Host
```

creates a `Base_Object + Template` combination. The object scheduling mechanism is overridden with an explicit instruction on where to place each combination, thereby ensuring failure independence from the other K replicates. Although, in

```
CLASS Resilience_Template(Base_Object : AnyObject);

   BEGIN
   VARIABLE Instance_List IS SET OF Object_Pointer;
   VARIABLE Instance      IS Object_Pointer;
   VARIABLE Copy          IS INTEGER;
   VARIABLE Result        IS Parameter_List;


   PROCEDURE Service_Initialisation(Resilience_Level : INTEGER);

      BEGIN
      VARIABLE Host IS Host_Name;

      Instance_List := [Base_Object];      /* initial instance */
                                           /* already created  */
      FOR Copy := 1 to Resilience_Level

        /* create k additional object-template pairs,          */
        /* placing template 'names' in the list of replicates */

         DO BEGIN
         Host := Find_Independent_Host(Instance_List);
         Instance_List := Instance_List +
            [NEW Resilience_Template(NEW Base_Object) @ Host];
         END of FOR

      FOR EACH Instance IN Instance_List
         DO Instance.Set_Cohorts(Instance_List);

      END of Service_Initialisation;


   PROCEDURE Set_Cohorts(Cohort_List : SET OF Object_Pointer);

      BEGIN
      Instance_List := Cohort_List;

      END of Set_Cohorts;
```

Figure 8–3: (Part 1) An Example Resilience Template Implementation

```
   FUNCTION Service_Provision( Service_Name       : Logical_Name;
                               Service_Parameters : Parameter_List );

      BEGIN
      FOR EACH Instance IN Instance_List
         DO Result := INVOKE Instance.Service_Update(Service_Name,
                                                     Service_Parameters);

      RETURN Result;     /* could use majority voting here */

      END of Service_Provision;


   FUNCTION Service_Update(Service_Name       : Logical_Name;
                           Service_Parameters : Parameter_List );

      BEGIN
      Result := INVOKE Base_Object.Service_Name WITH Service_Parameters;

      RETURN Result;

      END of Service_Update;


   PROCEDURE Service_Closedown;

      BEGIN
      FOR EACH Instance IN Instance_List EXCEPT Base_Object
         DO Instance.Service_Closedown;

      TERMINATE;
      END of Service_Closedown;

END of CLASS Resilience_Template;
```

Figure 8–3: (Part 2) An Example Resilience Template Implementation

principle, the template and base object do not *have* to be at the same location, in practice this configuration reduces communication overheads. The pointer to each `NEW Resilience Template` is noted for use later. Having established the multiple templates, `Service Initialisation` then informs each one, through an invocation on `Set Cohorts`, the names of its peers. Once each template has received the list of peers it is ready to participate in providing a resilient service.

Service Provision

The function `Service_Provision` receives all invocations intended for the base object. This is where the ability to substitute different object implementations becomes important. The *template* is seen as providing the service, and in order to use the service, clients must make invocations upon the template. Only the templates are aware of the base objects' existence.

When an instance is invoked, the service provision routine informs the other instances in its cohort list by invoking `Instance.Service_Update`. Each template thus informed invokes its associated base object, thereby keeping them (almost) consistent. In this simple example no attempt is made to ensure a strict cohort-wide ordering on invocation updates. Suitable synchronisation protocols are available, for example, Herlihy's quorum-consensus replication method [Herli86], but the details of their implementation are beyond the scope of this example. It is also possible to perform error detection and correction at this point, for example, by placing a timeout on each reply message. In this mock implementation, the results generated by each instance are assumed identical. The cohort results are therefore not examined for consistency. If required, a suitable 'majority voting' protocol could be used to determine the collective response. Having received the cohort responses the template then returns the result to the client.

Service Closedown

As with initialisation, service closedown is performed by invoking the template which then uses the system's (de)scheduling services to remove the multiple instances. Service closedown may be initiated from several sources, for example, by the original service requester, by the system scheduling mechanism, or during garbage collection. The algorithm shown here is inefficient since every one of the $K+1$ copies generates K 'shutdown' messages. However, it suffices to demonstrate the principle of using the template to remove the virtual property.

8.4.2 Persistence

Figure 8–4 shows a mock implementation of a persistence template. The `Service Initialisation` routine requires some means of checking for an existing service history from a previous (failed) incarnation. This is represented in the example by the `Recovery_Code` variable, which uniquely identifies a persistent object's invocation log (held in stable store). Should such a log exist, then the initialisation routine replays its contents to the new instance, thereby making its state identical to that of its failed predecessor. Subsequent invocations will be appended to this existing log. If no service log exists, then this is a new incarnation and consequently a new log must be created. Responsibility for assigning recovery codes to services is assumed to lie outside the template, i.e., with the environment in which the template is operating (see chapter 9 for further discussion on object environments).

During service provision the template copies the details of all invocations to the stable storage service log. Logging the invocation result, although not strictly necessary, allows the initialisation phase, in the event of a re-incarnation, to verify a correct replay sequence by comparing the new instance's responses with those of

```
CLASS Persistence_Template(Base_Object : AnyObject);

   PROCEDURE Service_Initialisation(Recovery_Code : UniqueCode);

      BEGIN
      If {existing log attached to this recovery code }
         THEN {open log and replay invocation history}
         ELSE OPEN_NEW(Service_Log) WITH Recovery_Code;

      END of Service_Initialisation;


   FUNCTION Service_Provision( Service_Name       : Logical_Name;
                               Service_Parameters : Parameter_List );

      BEGIN
      VARIABLE Result IS Parameter_List;

      Result := INVOKE Base_Object.Service_Name WITH Service_Parameters;

      LOG_TO(Service_Log, Service_Name, Service_Parameters, Result);

      RETURN Result;

      END of Service_Provision;


   PROCEDURE Service_Closedown;

      BEGIN
      DELETE(ServiceLog);  /* since no longer needed */

      TERMINATE;
      END of Service_Closedown;

END of CLASS Persistence_Template;
```

Figure 8–4: An Example Persistence Template Implementation

its predecessor. During service closedown, i.e., when terminating the persistence property, the service log can be deleted as it is no longer required.

8.4.3 Access Control

Figure 8–5 shows a mock implementation of a general access control template. There are no obvious initialisation or closedown actions required in this general example, so the corresponding routines have been omitted. Upon each invocation, the security parameters supplied by the client are validated appropriately. For example, a capability based template would receive capabilities, while an ACL based template would receive client names that it can check against an access control list (establishing the list members would be an initialisation task). Regardless of the security mechanism employed, if the security parameters are validated successfully then the base object is invoked (without security parameters). Invocations failing verification are rejected by the template.

Non-circumvention of the template can be assured, in the simplest case, by having only the template aware of the base object's existence. When this is infeasible or considered too insecure, more elaborate schemes can be employed involving the cooperation of (trusted) system software to ensure that only security templates may invoke secure services. If this is also infeasible or insecure, then access control using templates might be restricted to the role of *converting* between access methods. For example, if an object expects clients to present valid capabilities, then a template can be applied to allow ACL based clients to invoke this service. The template would validate the client against its authorisation list, forwarding successful invocation to the base object with a valid capability substituted for the the ACL security information. Conversions between other security mechanisms could be performed in a similar manner.

```
CLASS Security_Template(Base_Object : AnyObject);

   FUNCTION Service_Provision( Service_Name        : Logical_Name;
                               Service_Parameters  : Parameter_List;
                               Security_Parameters : Security_Info );

      BEGIN
      VARIABLE Result IS Parameter_List;

      IF VALID(Security_Parameters, Base_Object)
         THEN BEGIN
         Result := INVOKE Base_Object.Service_Name WITH Service_Parameters;
         RETURN Result;

         END of IF
         ELSE REJECT(Security_Parameters);

      END of Service_Provision;

END of CLASS Security_Template;
```

Figure 8–5: An Example Access Control Template Implementation

Using security templates it becomes possible to create several instances of the same service, each one protected by a different security mechanism. Such an environment provides greater flexibility in matching its services to clients' requirements.

8.4.4 Debugging

The implementation details of an inter-object debugger are beyond the scope of the simple examples presented here. However, following the same format as the other examples, the `Service_Provision` routine, which observes all invocations and results, is able to provide the inter-object debugging facilities identified earlier : recording and displaying invocation behaviour, parameters and results; storing invocation histories — perhaps in the same manner as a persistence template —

to be replayed later in order to reproduce error conditions; allowing modification of parameters and results.

8.4.5 Performance Monitor

An example implementation of a performance monitor template is shown in Figure 8–6. It is principally intended for the update thresholding mechanism developed in chapter 7, although it also applies to any update mechanism based upon observing invocation service times. The template relies upon its host to time-stamp incoming invocation messages as they are placed in its request queue. The model of invocation remains the same as described in chapter 6 (section 6.1), except that message queueing now occurs within the template object rather than the base object.

The template keeps a list of objects wishing to receive updates relating to the base object. Any object wishing to receive updates adds its name to the list by invoking `Add_Monitor`. In principle there is no upper limit on the number of monitors.

Invocations upon the base object are time-stamped and placed in the template's request queue to await servicing. It is assumed that the time-stamp information can be extracted by the template for use in calculating the invocation service time. The `Service Provision` routine services each request in turn, removing it from the request queue and passing it to the base object. Upon completion of the invocation, the template obtains the current time from its host, i.e., the end-time for the complete invocation, and uses this to calculate the service-time. This service-time includes the message queueing time within the template and the actual invocation service-time by the base object. Having updated its service history with this latest observation, the template then calls its update algorithm (in this case the thresholding mechanism). If this algorithm decides that

```
CLASS Monitor_Template(Base_Object : AnyObject);

   BEGIN
   VARIABLE Monitor_List IS SET OF Object_Pointer;

   PROCEDURE Service_Initialisation;

      BEGIN
      Monitor_List := [];

      END of Service_Initialisation;


   PROCEDURE Add_Monitor(New_Monitor : Object_Pointer);

      BEGIN
      Monitor_List := Monitor_List + [New_Monitor];

      END of Add_Monitor;


   FUNCTION Service_Provision( Service_Name       : Logical_Name;
                               Service_Parameters : Parameter_List );
      BEGIN
      VARIABLE Start_Time   IS Time_Details;
      VARIABLE End_Time     IS Time_Details;
      VARIABLE Service_Time IS REAL;
      VARIABLE Result       IS Parameter_List;
      VARIABLE Monitor      IS Object_Pointer;

      EXTRACT_TIME(Start_Time);    /* from invocation message, as   */
                                   /* recorded by template host     */
                                   /* upon message's insertion into */
                                   /* template's request queue      */

      Result := INVOKE Base_Object.Service_Name WITH Service_Parameters;

      GET_TIME(End_Time);          /* current time, from host's clock */

      Service_Time := End_Time - Start_Time;

      {update rolling service history with Service_Time}

      IF {update required according to thresholding algorithm}
         THEN FOR EACH Monitor IN Monitor_List
            DO Monitor.Update(Current_Average, Service_History_Length);

      RETURN Result;

      END of Service_Provision;

END of CLASS Monitor_Template;
```

Figure 8–6: An Example Performance Monitor Implementation

an update *is* required, then each object in the monitor list is invoked to receive the current performance information. If no updates are required then no action is taken. Finally, the invocation result is returned to the client.

8.5 Implementation Issues

Having illustrated the principle behind virtual templates, this section examines some problems associated with implementation.

8.5.1 Template Performance

The mock implementations described above assume that virtual templates are independent objects, separate from the base objects they support. Although this approach is feasible, it is potentially inefficient due to the additional invocations generated by the client invoking the template, which then invokes the base object (Figure 8–2). A similar overhead is incurred when returning results since the base object replies to the template, which then replies to the client. The benefit of adding a virtual property may compensate for a *limited* performance degradation; however, doubling the amount of inter-object communication may not be acceptable. One possibility for reducing this overhead is to always place the template at the same location as the base object, since intra-location (intra-host) invocation is generally less expensive than inter-location invocation. Extending this idea further, a more efficient approach would be to incorporate the virtual template as part of the base object, thereby removing the additional invocation overhead. The object paradigm provides a possible solution to this integration through the use of inheritance.

There are at least two approaches to incorporating virtual templates using inheritance; one of these implements templates as the generic concept described

here, while the other provides the programmer with a 'toolkit' for creating service-specific virtual properties. Both these approaches are examined below.

8.5.2 Inheriting From Base Objects

By placing a template class description at the bottom of a class hierarchy (Figure 8–7), the template can inherit, using standard object oriented inheritance mechanisms, the interface and invocation routines of all its ancestors; in particular, it inherits the interface and invocation routines of the base object. In effect, the template extends the base object's interface to include routines such as `Service Initialisation`, `Service Provision` and `Service Closedown`. The main requirement for creating virtual properties in this way is for the `Service Provision` routine to intercept all invocations upon the base object's services, which requires the cooperation of the underlying invocation mechanism.

Objects known to regularly require certain virtual properties can be constructed in this manner, with the appropriate templates included. Ideally, the construction service should append template classes automatically while constructing the object. This hides the virtual property implementation entirely from the base object programmer, which is in keeping with the original template concept. It also allows different users to specify different properties throughout the base object's life-time.

Implementing virtual properties using base object inheritance does not interfere with the original, separate template object approach; the two techniques may co-exist. This maintains the flexibility provided by independent templates, whilst also offering 'good' performance by enabling objects to be created with 'built in' virtual properties.

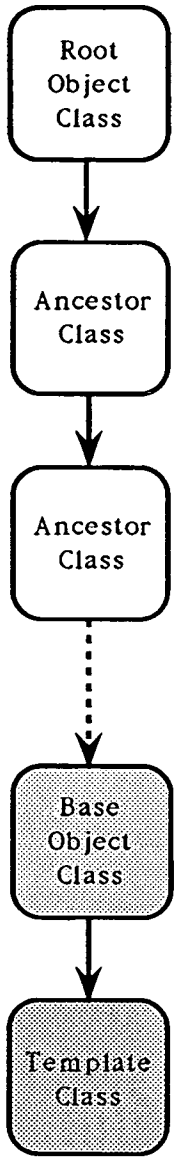

Figure 8–7: Template Classes Inheriting from Base Object Classes

8.5.3 Inheriting From Templates

If a template class is included near the root of the class hierarchy (Figure 8-8), then all user defined classes (which are now sub-classes of the template class), automatically inherit the virtual property invocation routines. Hence, *any* user defined object can be invoked with a request such as `Resilience_Initialise(3)`, the code for which will be found in the object's class hierarchy using the standard inheritance mechanisms.

The problem with this approach is in redirecting invocations via the `Service Provision` routine, since the base object is (notionally) unaware of its existence. Requiring the base object to invoke `Service Provision` explicitly upon each invocation removes the 'hidden' element from the template concept. Although not strictly an implementation technique for templates, if this approach is taken, a service-specific virtual property mechanism can be introduced.

The Arjuna system [Shriv88] uses object oriented inheritance to provide persistence, recoverability[1] and concurrency control [Dixon88] [Parri88]. The root class `Object` provides the basic facilities allowing a type to be recoverable and persistent. Further sub-classes[2] of `Object` build upon these basic facilities to provide atomic actions and locking facilities for concurrency control, which are then automatically inherited by new classes.

In general, the template classes defined near the root of the class hierarchy should provide only the low-level building blocks required to construct their respective virtual properties. Further sub-classes inherit these facilities and may build upon them to create 'better' and 'larger' facilities. These facilities are then

---

[1] Recoverability is the ability to 'undo' a series of invocations.

[2] *All* classes are, by definition, sub-classes of `Object`

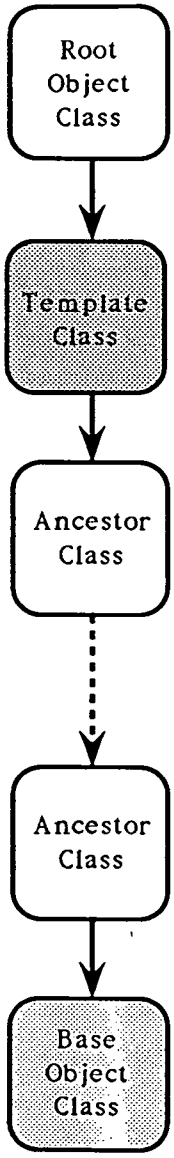

Figure 8–8: Base Object Classes Inheriting from Template Classes

automatically inherited by 'application' objects, which can call upon them explicitly to realise a particular virtual property. However, object programmers also have the freedom to enhance or even re-define these facilities using standard object oriented programming techniques. Hence, the virtual property can, if necessary, be customised for the object in question while still re-using inherited code that the object's programmer need not reproduce, or indeed understand (beyond the interface level).

8.6 Summary

A particular group of virtual properties have been identified, each of which can be created by manipulating invocation messages and base object instances. It has been argued that the object and invocation manipulation required to create each of these properties can be encapsulated in a generic, re-usable form; namely virtual templates. Several example templates have been shown, covering Resilience, Persistence, Access Controls, Debugging and Performance Monitoring. Conceptually, templates are independent objects interposed between client and server. However, performance issues make it desirable to incorporate the template within the service object. Two possible approaches were identified using inheritance : template classes inheriting from base object classes to provide 'hidden' virtual property implementation; and base object classes inheriting from template classes, enabling programmers to build upon and customize, service-specific virtual properties.

Chapter 9

Resource Provision

This chapter examines how the various aspects of resource provision may be combined to provide a complete resource scheduling mechanism. The early part of the chapter speculates on how users might access the distributed system's resources using Environments, in which only virtual objects exist, with no notion of location or distribution. Based on this user oriented view, the resource provision requirements associated with various user actions are determined. In each case, a solution is offered utilising the techniques developed in this thesis. The chapter concludes by examining the limitations of these mechanisms, suggesting the circumstances under which resource provision techniques can be expected to operate effectively.

9.1 Environments

Before examining resource provision in detail, this section speculates on how the resources of a distributed system might be presented to users. Within this framework, several user actions are identified that have an effect upon resource provision. These actions are used later as a basis for describing how the resource provision requirements can be satisfied using the techniques developed in this thesis.

In large multi-user, object oriented distributed systems, individual users are normally interested in only a relatively small subset of the many thousands of objects available. There is therefore a need to structure the user's view of the system in order to sensibly organise and find objects and services. This is not unlike the problem of structuring multi-user file services in non-object based systems. Each user wishes to see only a small subset of the entire system, containing only those objects of interest.

An **Environment** is a restricted view of the virtual object world customised by user and by activity. Each environment contains only those objects of interest to a specific user when performing a particular 'flavour' of task. From the user's point of view, each environment is a self-contained world providing all the facilities required to perform a specific task. The user perceives only virtual objects in the environment, with no concept of location or distribution. Objects can be added to or removed from an environment at will, with all scheduling and resource provision being performed automatically and invisibly.

In [Neuma89], Neuman describes a similar approach to user environments, known as the Virtual System Model, based upon the observation that large systems are difficult to manage and negotiate, and that users should be presented with a small subset of the system containing only those parts of interest.

> "The Virtual System Model provides a framework within which users can build a view of a system in which the parts of interest are logically nearby." [Neuma89]

The concept is illustrated using a virtual file system implementation in which the user defined directory hierarchy is independent of the underlying storage hierarchy. Files can appear many times within a directory hierarchy, as well as in multiple hierarchies. The directory hierarchy may also contain loops. Some (essential) files appear in all users' hierarchies, but in general each user has a customised view of the file system.

9.1.1 Environment Examples

While using a workstation, a user may have several (screen) windows open, each containing a different environment. Examples are numerous, but could include environments such as : SmallTalk, perhaps a distributed implementation; a documentation environment, complete with the user's favourite text processor, a dictionary object, a thesaurus object, printer objects and the document objects themselves; programming language development environments for creating new object classes, including language manuals and programming tools (note that in a user environment, compilation will be performed automatically. The programmer is aware of only one representation of the object, which is the source code representation); operating system environments providing emulations of specific systems, perhaps non-object oriented and non-distributed; a system management environment providing system configuration tools and privileged access to system data. In a mature system there will be many other environments, created and customised by the users themselves. As an example, application programs may have execution environments created for them in which all the facilities required by the application are (logically) collected together.

9.1.2 Hierarchies

In an object oriented system the environments will themselves be implemented as objects, and can therefore appear as members of other environments. Hence, a hierarchy of environments can be established such as the example shown in figure 9–1. The root of the environment hierarchy, `SWP`, is established automatically when user SWP accesses the system. As shown here, there are two 'sub-environments' in the `SWP` environment : `SWP_Documents` and `SWP_Simula`. The `SWP_Documents` environment contains several document objects created by user `SWP`, represented here by the objects `Thesis` and `Paper`. The document environment also contains the sub-environment `Document_Tools`, which contains objects such as a text editor, dictionary and thesaurus. This in turn has access to a `Printers` environment, which contains printer objects suitable for producing hard copies of documents. The `SWP_Simula` environment, which contains program objects, has a similar environment hierarchy. The `Simula_Tools` environment has a language manual object, an editor object and access to a `Printers` environment, although this may not be the same as the documentation `Printers` environment since it may contain different printer objects.

Each environment may provide its own aliases for the objects it contains. Objects in other environments can be accessed using an `Environment:LocalName` pair. Note that an object may be present in more than one environment and can therefore be referenced by more than one name.

Note that all the objects mentioned here, and indeed the environments themselves, are *virtual* objects, operating at the level of abstraction defined by the uppermost layer of the Object Reference Model (see Chapter 2).

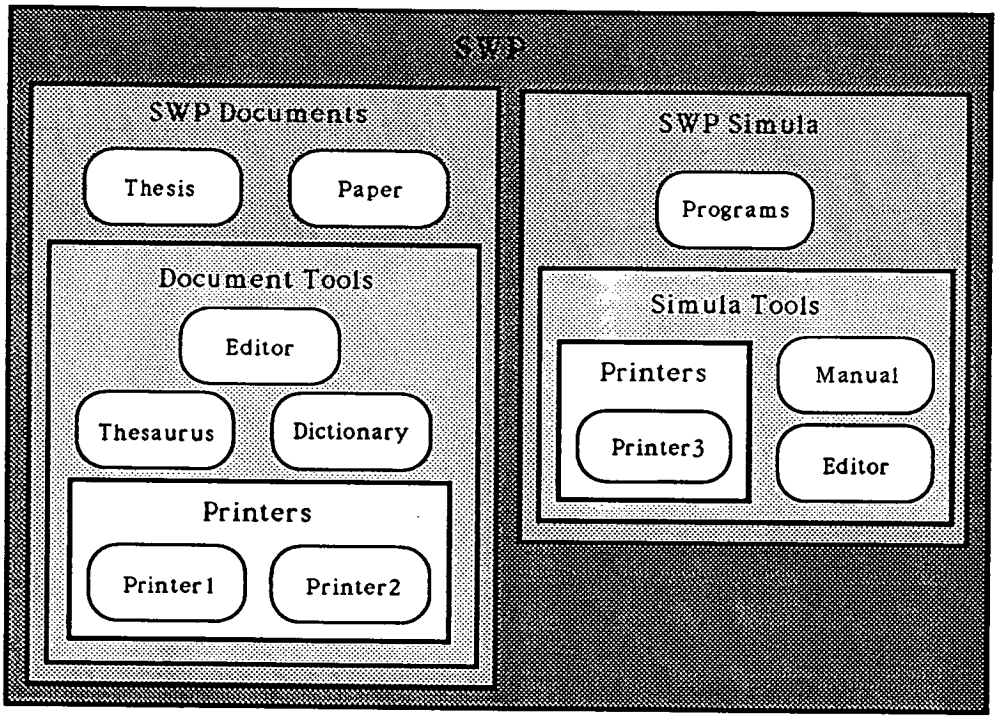

Figure 9–1: An Example Environment Hierarchy for User SWP

9.1.3 User Interface

Although not directly relevant to the issues under discussion, a brief examination of how environments might be presented to users, and the actions users can perform within them, provides some insight into the resource provision issues that must be tackled.

As stated in chapter 3, the system is assumed to provide high resolution icon, window and mouse based user interfaces. Objects and environments can therefore be presented as icons. Using the mouse pointer to select an object icon may perform one of several actions such as displaying user oriented information relating to the object's function, deleting the object, or listing the object's invocation routines. Selecting a routine name invokes the object. Selecting an environment icon 'opens' the environment, creating a new window containing its associated object (and environment) icons. Objects could be copied between environments simply by 'dragging' their icons from a source environment window to the destination environment window. A separate mechanism must be provided to enable users to find, and hence access, unknown environments (*cf.* the `directory` command used in file systems). Users of 'programming' environments require a further interaction mechanism to enable new object classes to be incorporated into the system (the details of such a mechanism are discussed later).

Hence, at the environment level, there are three main activities a user can perform that affect resource provision : object addition, object invocation and object removal. The following section examines the actions required of the resource provision mechanisms in each case. Initially, only simple environments are considered, with no sharing of objects. Sharing is examined later.

9.2 Resource Provision

The descriptions of resource provision services given below are based upon the operational services described in chapter 2 for each of the ORM layers. The mechanisms identified for implementing these services are based upon the work presented throughout the thesis. The intention is to relate the various research aspects that have been developed, indicating how they might combine to provide a complete resource provision facility. In common with the rest of the thesis, the following discussion is not intended to provide an implementation description, but rather indicates the general techniques that could be used.

9.2.1 Configuration Management

Environments create small, self contained systems utilising a subset of the resources provided by the larger, underlying distributed system. They provide structure to an otherwise flat virtual world. The grouping of services into environments can be configured at all levels of the resource provision hierarchy, assisted, for example, by using an `option` parameter to pass environment identifiers down through the resource provision hierarchy. Restrictions and controls can therefore be placed upon the resources available to individual environments. For example : at the Virtual Layer, restricting the availability of virtual properties; at the Invocation Layer, restricting the level of service expansion and contraction permitted; at the Location Layer, limiting the number of candidate locations when scheduling new objects; at the Construction Layer, limiting the range of representation transformations available; and finally, at the Migration Layer, limiting the processors to which access is permitted.

The scale of resource scheduling can therefore be reduced from one large environment to many smaller environments. The controls and restrictions established

at each level of the resource provision hierarchy should be hidden from the other layers. They might be established and maintained by the (human) system administrators using external-management (configuration) services provided for this purpose. Hence, at each level in the resource provision hierarchy it should be possible to configure resource provision to suit the users, their applications and the underlying system.

9.2.2 Adding Objects to an Environment

Adding an object to an environment initially only requires placing the name of the (virtual) object in the environment's list of known objects. At this point an icon can be added to the user's display, indicating that the object is available. No further resource provision activities are required until the user attempts to invoke the object. When this occurs, the actions of the resource provision services depend upon the nature of the added object; it may be new to the system, new to all currently 'active' environments, or simply new to *this* environment.

If the (virtual) object is new to the entire system then it must be incorporated into the 'list' of known objects. Section 9.2.3 examines how this might be achieved. If no scheduled instance of the object currently exists, then a new instance must be created, possibly using the techniques described below in section 9.2.4. If the object exists in another environment, the 'new' virtual object may possibly be mapped onto this existing object. Alternatively, if the existing object is unable to support an additional client, a new instance must be scheduled.

9.2.3 Adding New Objects to the System

New object classes are added to the system by making them available to the appropriate type representative. Classes under development should be identified as 'text' objects rather than class descriptions, in order to avoid the construction

facility automatically picking up incomplete classes. Hence, they can be read, edited, printed and so on, in the same manner as other document objects. For example, in a `C++` environment the `C++` programmer would create a module containing C++ code to perform the services required of the new object. For this part of the programming process the newly developed code should be treated exactly as a document. Once completed, the programmer changes the object's type from `Document` to, say, `C++Module`. The environment must vet each new object for compliance with its language definition, i.e., type checking it for conformance with its purported type and registering it with the appropriate type representative. Once the environment has verified the type conversion, the object can no longer be read, edited or printed, as these are invocations made upon documents. Selecting the object now gives a different list of possible invocation routines, namely those defined by the programmer in the body of the `C++` module.

The verification aspects of object construction have not been tackled in this thesis. Type verification should sensibly be a function of the construction service, which encapsulates all knowledge of object representation. Programmers should remain unaware of verification activities except in the event of an incompatibility, such as a syntax error. Hence, programmers perceive only a new object in their 'programming' environment, which they are now free to invoke in the usual manner, possibly requesting the virtual property 'debuggability' (see Chapter 8) in order to test the object's behaviour.

9.2.4 Scheduling New Objects

When the very first invocation is made upon an object, the resource provision mechanisms are called upon to create an executable instance (Figure 9–2). In order to create a virtual object the environment calls the Virtual Layer `CreateService` facility, specifying the virtual properties required. This service, which can be embodied within a 'virtual world management' object (hidden from the environ-

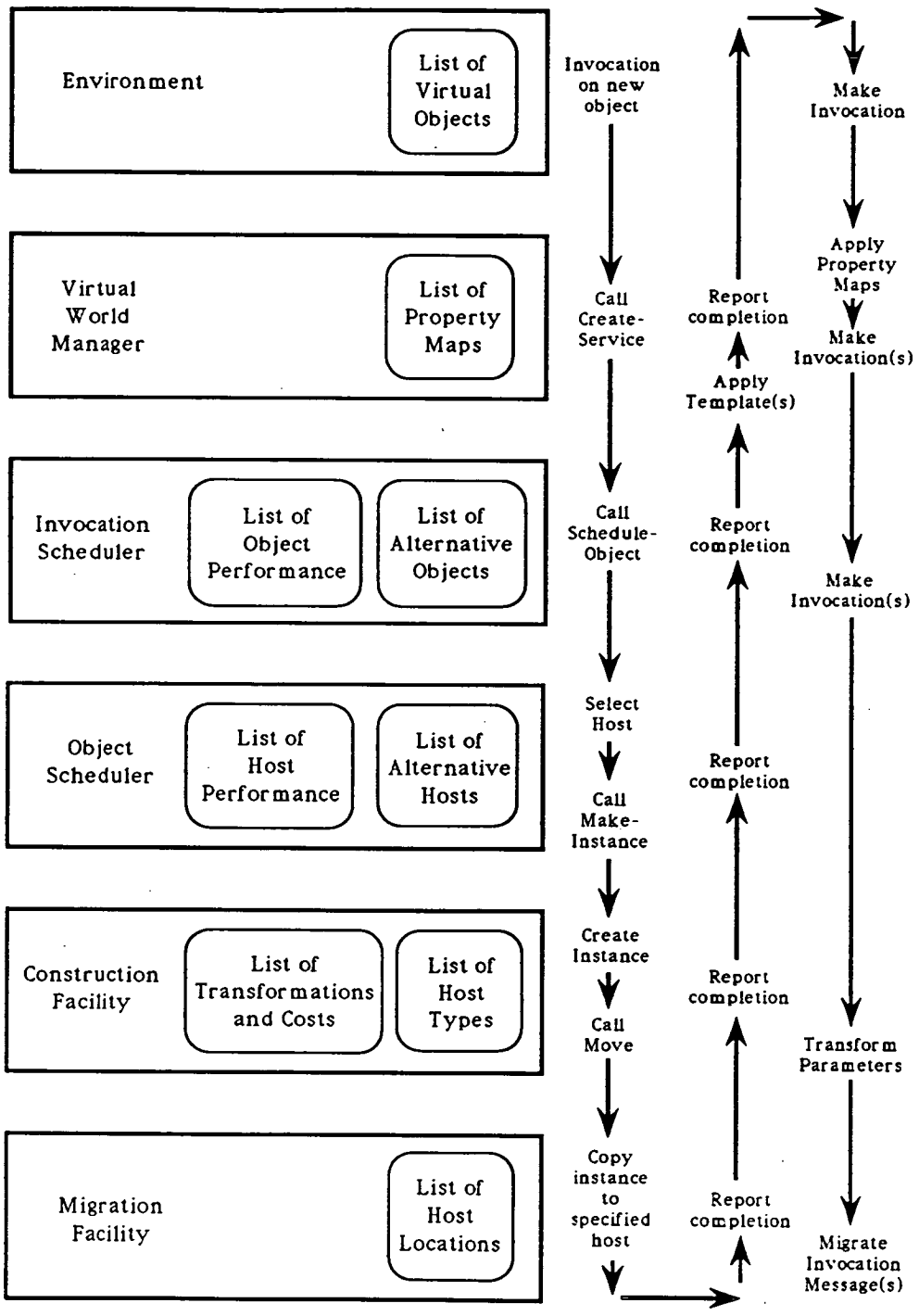

Figure 9–2: The Resource Provision Hierarchy

ment's users) in turn calls the `CreateService` facility embodied by the Invocation Layer. The Invocation Layer services then call the `Schedule` facility offered by the Location Layer's object scheduler. It is here that the main scheduling activity takes place.

The object scheduler, which continually monitors the performance of each object host available to the originating environment, selects the 'best' host to receive the new instance. This decision is made in conjunction with the management information provided by the construction service, which defines the possible representations available and their relative transformation costs. Chapter 6 (section 6.7) discussed the use of comparison scheduling in selecting a suitable host. The update thresholding mechanism presented in chapter 7 can be used to reduce the update traffic describing each host's current performance.

Having established the target host, the object scheduler calls upon the object construction facility to `MakeInstance`, which can use the construction algorithm described in chapter 4 to create an appropriate object representation. Finally, the construction facility calls upon the migration service to install the object representation on the target host. As each scheduling stage completes, it reports back to the previous level. When the completion message reaches the virtual world manager, the management service invokes the appropriate initialisation routines for each of the virtual properties specified by the user. Object installation is now complete, and the invocation that triggered this process can proceed as normal (see below). Figure 9–3 shows the mapping that has been established; from the abstract environment object onto an executable representation resident on a processor.

9.2.5 Invoking Objects

Invocations upon the objects in an environment are mapped to the appropriate virtual object and passed to the virtual world 'manager' (see rightmost path in

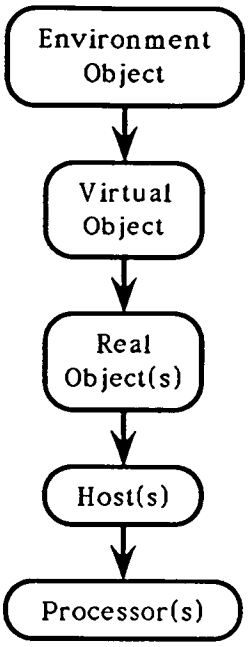

Figure 9–3: Mapping an Environment Object onto a Processor

Figure 9–2). The virtual manager applies its 'rules' for maintaining virtual properties, converting the virtual invocation into the appropriate real invocation(s). The use of virtual templates for performing this task was discussed in chapter 8. The real invocations are then passed to the invocation scheduler for assignment to real objects.

As well as assigning invocations to objects, the invocation scheduler is also responsible for controlling service expansion (and contraction) to maintain a 'satisfactory' level of service. The thresholding technique can be used to detect when an object's average service time falls below an 'acceptable' level (see section 7.4 for more details). Under these circumstances, the object scheduler is triggered to create an additional object instance, using the same techniques as described above. New instances are added in this manner whenever the existing instances become overloaded. In the case of retentive services, the invocation scheduler is

further responsible for instigating the transfer of status information between the original and new service instances (see chapter 5).

Note that service expansion is a function of the invocation scheduler, and remains invisible to the virtual manager, which perceives only a single instance of each real object associated with the virtual object. If a service has been expanded, i.e., with multiple real object instances providing identical services, then comparison scheduling, described in detail in chapter 6, can be used to balance the invocation load.

Having made the invocation scheduling decision, the invocation message(s) must be forwarded to the appropriate object(s). Notionally, this does not involve the object scheduler, although some address space translation may be required for which the object scheduler must be consulted. In particular, the object scheduler may be performing object migration to compensate for coarse-grained load imbalances between hosts. As with the service expansion provided by the invocation scheduler, object migration for load balancing is a 'hidden' function of the object scheduler. The invocation scheduler perceives only scheduled objects, with no attached notion of physical location.

Finally, if necessary, the construction services may transform the message parameters into a representation accepted by the target object. The migration service is then called upon to deliver the invocation message(s) to the object(s) specified by the invocation scheduler.

9.2.6 Sharing Objects

Object sharing may occur at several places in the resource provision hierarchy. Figure 9–4 illustrates several possibilities. The environments are responsible for establishing and maintaining the sharing of virtual objects. Some virtual objects, although apparently independent to users, may in fact share the same real object

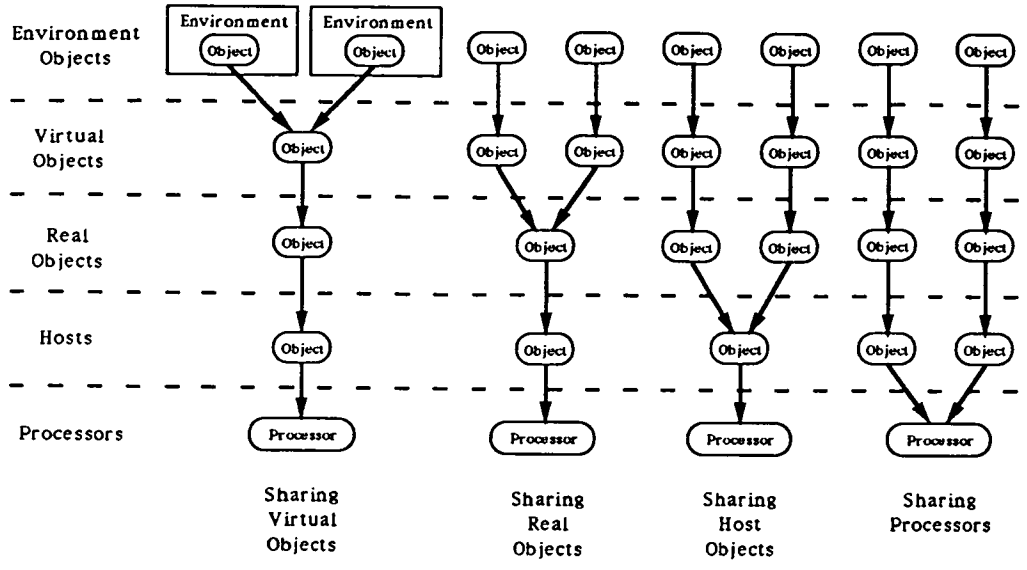

Figure 9–4: Sharing Objects

instance(s). As an example, most environments will include some sort of name server or directory object to identify other objects and environments. Each environment may appear to have its own copy of this service, although in reality they could all map to a single 'real' service. The virtual manager is responsible for establishing this level of sharing.

Multiple real objects may share a host object. This is one of the basic assumptions stated in chapter 3, that most hosts are capable of supporting more than one object instance. The object scheduler is responsible for assigning objects to hosts, attempting to share the workload evenly. Finally, object hosts may share processors. The mapping of hosts to processors is likely to be static, pre-determined by the (human) system administrators.

Figure 9–5 combines some of these sharing modes to provide a more realistic mapping of environment objects onto physical nodes. The figure shows a portion of two environments; `Environment 1` and `Environment 2`. Both environments have a `Directory` object enabling users to find, and hence use, objects in other environments. Although seen as independent services by the environment users, both

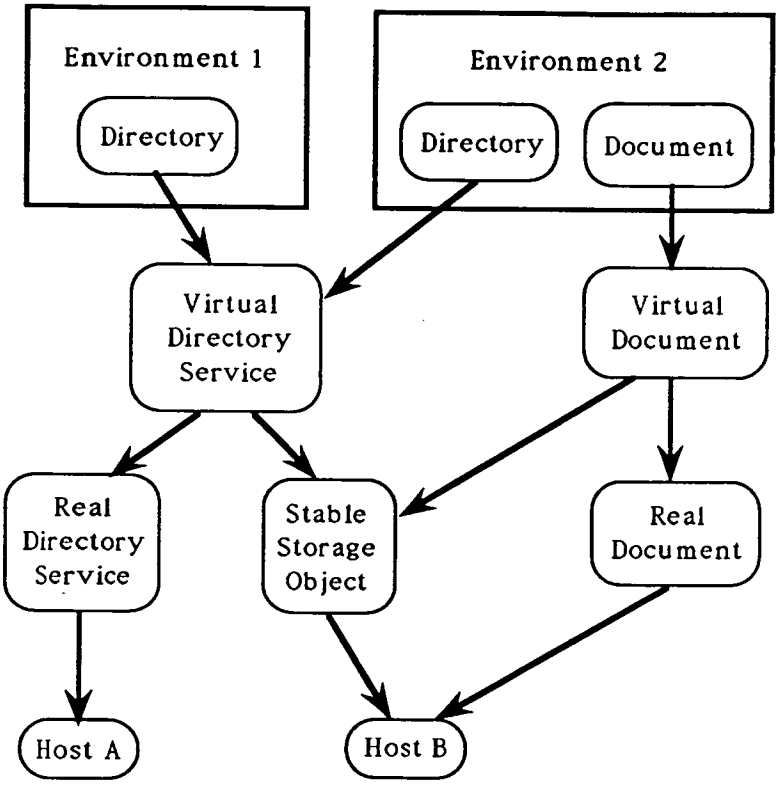

Figure 9–5: An example Sharing Configuration

`Directory` objects share the same virtual object. In the example, a persistence template has been applied, so the virtual directory object is realised using two real objects — the directory server object and a stable storage object — which reside on host A and host B respectively. As well as a `Directory` object, `Environment 2` contains a `Document` object which, like most document objects, must be persistent in order to retain the text of the document between work sessions. The virtual document object is therefore realised using the 'real' document object and a stable storage object, which in this example is the same stable storage object as used by the directory service. Finally, both the document object and the storage object share host B.

9.2.7 Object Deletion

At the environment level, removing an object simply involves deleting the object's name from the environment's list of known objects, in conjunction with removing its associated icon. How far this deletion command is propagated down the resource provision hierarchy depends upon the scheduling policies being used and the sharing configuration. In each case, the decision to de-schedule, thereby freeing potentially valuable resources, must be weighed against the likelihood, and cost, of the object being reinstated in the near future. An object can only be de-scheduled if it is not being shared with any other service.

The virtual world manager de-schedules a virtual object by requesting the invocation scheduler to remove the components of the 'real' service. Hence, using the example in Figure 9–5, in order to remove the virtual document object the invocation scheduler would be requested to delete the stable storage object and the real document object. It does this by calling upon the object scheduler to `DeSchedule` the real objects. In this simple example there is a direct correspondence between the real objects known to the virtual world manager and those used to implement the service. In other cases the invocation scheduler may have expanded the service by creating additional instances. Under these circumstances the invocation scheduler will instruct the object scheduler to delete *all* instances associated with the virtual service. Finally, the object scheduler informs the construction facility, which may then delete the object's executable representation.

9.3 Limitations

This section examines the effects upon the scheduling mechanisms of user behaviour and system configuration. For each scheduling aspect, the extremes of operation are examined, with corresponding user actions or system configurations

identified that lead to these conditions. The resultant discussion defines the circumstances under which these resource scheduling mechanisms are expected to operate effectively.

9.3.1 Virtual Properties

The application of virtual templates to objects generally incurs some sort of overhead, ranging from increased communication to the creation and maintenance of additional object instances. Under normal circumstances this cost is an acceptable consequence of the virtual objects' increased utility. However, users requesting multiple properties on every object may generate considerable overheads, resulting in reduced performance. Virtual properties should therefore only be applied to objects when necessary. In particular, properties that generate multiple instances, such as resilience, should not be applied indiscriminately. The system may have to impose limits on users of these properties, for example, placing a sensible maximum on the level of resilience permitted, to avoid requests for '1000-resilient' objects. Objects possessing no virtual properties[1] do not create any additional overheads.

9.3.2 Invocation Rates

Although not under direct user control, the invocation rate is influenced by user actions since user requests initiate invocation activity. During slow periods, a single user may not generate sufficient work to justify exclusive access to a dedicated object. Under these circumstances the invocation scheduler's ability to share objects between users becomes important. Where possible, users should be grouped

---

[1]Notionally, all objects possess at least a 'null' virtual property, since only virtual objects can inhabit environments

together to be collectively served by a single service instance. The mechanisms presented here do not tackle this problem.

At high invocation rates, the invocation scheduler may expand overloaded services by creating additional instances to share the load. These additional instances are removed when the invocation rate falls. However, bursty invocation activity, i.e., alternating periods of high and low invocation rates, may cause the continuous expansion and contraction of services. It has not yet been established whether thresholding (see section 7.4) can successfully avoid this sort of 'service thrashing', but the experience with the thresholding of update messages suggests that it may be suitable.

9.3.3 Object Scheduling

The principal constraint on an object scheduler is the number and type of hosts available to it. The earlier discussion on configuration management suggested that each environment may be associated with an arbitrary group of hosts. Obviously, if the scheduler has no hosts then it cannot schedule any objects. If only one host is available then the scheduling problem is trivial, although this could be a typical configuration, with the only known host corresponding to the physical node owned by the user. Object scheduling becomes interesting when there are two or more hosts available. A greater number of hosts offers greater potential for useful load sharing, although this must be offset against the monitoring overhead. The 'optimal' sized grouping of hosts will vary between systems, depending upon the type of hosts involved, their capacity to support monitoring status information, and the level of system activity. Grouping hosts on a per environment basis provides a potentially flexible configuration mechanism, allowing the scale of object scheduling to be customised to the system under consideration.

9.3.4 Construction Facility

In a heterogeneous environment with many different host types, the construction facility will be called upon more heavily than in a largely homogeneous environment. For example, if every host is a different type, then an object has a high probability of being re-constructed every time it is scheduled. Conversely, if the majority of the host sites are homogeneous, then a representation created for one host can automatically be made available to the others through the type representative. Hence, greater homogeneity implies less construction overheads. These trade-offs are fundamental to any heterogeneous distributed system, independent of the construction mechanisms employed. Therefore, in general, automatic construction of objects is most suited to a heterogeneous environment in which each host type is replicated many times, or where one particular host type predominates.

9.4 Summary

This chapter has speculated on how distributed resources might be presented to system users. The concept of environments were introduced, where an environment is a self contained 'world' populated exclusively by virtual objects with desirable properties. Each environment is customised by user and by activity, (logically) collecting together the objects necessary for performing the activity. The resource provision aspects associated with realising these environments were explored. This discussion indicated how the resource provision techniques presented in this thesis can be utilised to provide a complete resource provision facility. Finally, some limitations of these techniques were identified.

Chapter 10

Conclusions

Using objects to structure distributed systems is becoming an increasingly popular paradigm. This thesis endorses the object approach. The reasons for this lie in the nature of the objects themselves; hidden 'black box' implementation (encapsulation), and communication by message passing. Other, non-object, paradigms also exhibit encapsulation and message passing, but they lack the other key object attribute of inheritance. This thesis has provided a study of resource provision in object oriented distributed systems. Several aspects of resource provision have been examined in detail, with object oriented solutions, mechanisms and recommendations being made as appropriate. It has been demonstrated that exploiting object attributes is a useful approach to accomplishing resource provision.

10.1 Thesis Summary

This research has studied resource provision in object oriented distributed systems. The main emphasis has been on demonstrating the value of developing object based solutions, rather than applying existing process based solutions. The introductory and background material was covered in chapters 1, 3 and 5. Chapter 1 introduced objects and distribution, including an overview of several example systems. Chapter 3 defined the target environment addressed by the thesis, while chapter 5 provided an overview of distributed scheduling. The key research contributions documented by the remainder of this thesis are summarised below.

10.1.1 The Object Reference Model

Although developed primarily as a framework to describe the work in this thesis, the Object Reference Model (ORM) is a general model of object oriented distributed systems that can usefully be applied to other areas of object oriented systems research. ORM, which was described in Chapter 2, provides a logical framework, built up as a series of layers, incorporating the various aspects of distributed systems design. Inspired by the OSI layered model of Communicating Systems, ORM is intended to assist the design of new systems whilst also allowing existing designs to be compared and contrasted.

10.1.2 Construction Graphs

A construction graph is a data structure that embodies the transformations performed upon a class definition in order to create an executable object instance. Construction graphs can be used to assist in the automatic construction and transformation of object representations in a heterogeneous environment. The concept

behind construction graphs was presented in chapter 4, along with the design of a distributed automatic-construction facility.

10.1.3 Comparison Scheduling

Comparison Scheduling applies statistical hypothesis testing techniques to the distributed scheduling of object invocations. When applied to invocation scheduling, an object instance is selected to receive an invocation only if it is 'significantly faster' than all other contenders; where 'significant' is defined in terms of the statistical significance of a hypothesis test. Chapter 6 described in detail one particular formulation of comparison scheduling, based upon the assumption of exponential service times. Simulated performance results compared favourably to those for random and greedy scheduling, with improvements of up to an order of magnitude observed. Based on these simulation results, a recommendation was made for level of statistical significance to be used when performing comparison scheduling.

10.1.4 Update Suppression

Chapter 7 presented an algorithm for eliminating redundant performance status update messages, based upon the same hypothesis testing techniques developed for comparison scheduling. The mechanism filters redundant messages, resulting in a 95% reduction in update communication with little or no corresponding reduction in scheduler performance. Used together, comparison scheduling and update thresholding offer considerable potential to improve scheduling performance in object oriented distributed systems.

10.1.5 Virtual Templates

Virtual properties and virtual objects were defined by the Object Reference Model described in chapter 2. Virtual templates encapsulate virtual properties, such as fault tolerance and persistence, in a generic, reusable form. In theory, *any* object instance can be made to display a virtual property simply by applying the appropriate template. Chapter 8 identified several example properties, providing mock template implementations for each one. Some of the more general problems associated with implementing templates were also discussed.

10.2 Future Work

10.2.1 Implementation

The various resource provision mechanisms presented in this thesis were developed without reference to any particular distributed system. This has the advantage of not restricting ideas to 'things that can be implemented with this particular box', thus encouraging solutions to fundamental, rather than system specific, problems. This approach, epitomised by the Object Reference Model, has been adopted throughout, with several different aspects of resource provision being addressed from a fundamental point of view. However, by using this approach, none of the ideas presented are proven by implementation. To do so would require an implementation effort beyond the scope of an individual Ph.D. project. There is therefore potential for further work in proving these ideas through implementation, either by incorporating them separately into suitable existing systems, or by designing and building a new system.

10.2.2 ORM

The formulation of ORM given in chapter 2 is relatively stable and has served the purpose for which it was intended; namely, to provide a descriptive framework for the research presented in this thesis. ORM is potentially of benefit to other researchers in this area, for structuring discussion about distributed systems. To this end, ORM could be further promoted as a self-contained model of object oriented distributed systems.

10.2.3 Comparison Scheduling and Status Updates

The simulations described in chapters 6 and 7 demonstrate the potential for comparison scheduling and update suppression to out-perform current scheduling techniques. However, further work should be performed to verify (or otherwise) that the assumption of exponential service times is reasonable across a wide range of object environments. Should this assumption not hold, then the hypothesis tests can be re-formulated using more suitable distributions. In principle, the re-formulated algorithms should perform exactly as the algorithm in chapter 6. The only drawback is that any non-exponential formulation will almost certainly lead to a more complicated test statistic, yielding a greater computation overhead. Further work could also be performed to establish the feasibility of using a comparison scheduler for assigning objects to hosts.

10.3 In Conclusion

This thesis has addressed some of the fundamental problems associated with resource provision in object oriented systems, attempting to provide object oriented solutions rather than applying existing, process based solutions. As a result, the

comparison scheduler and status update algorithms successfully exploit both encapsulation and message passing, while virtual templates may employ inheritance in order to create generic properties. In each case, it has been shown that utilising object oriented attributes, as opposed to process based attributes, provides useful solutions to resource provision problems. The conclusion reached is that when objects appear in distributed systems, the object paradigm should be used at *all* levels, because treating objects as processes ignores useful attributes that could otherwise be employed in controlling resource provision activities.

Bibliography

[Almes85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska and Jerre D. Noe. **The Eden System: A Technical Review.** *IEEE Transactions on Software Engineering*, Vol SE-11, pp. 43–58, January 1985.

[Almes87] Guy T. Almes. **Edmas: An Object-Oriented Locally Distributed Mail System.** *IEEE Transactions on Software Engineering*, Vol SE-13, No. 9, pp. 1001–1009, September 1987.

[ASN.1] **Open Systems Interconnection : Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).** *International Standards Organisation*, ISO DIS 8825, 1985.

[Barak85] Amnon Barak and Amnon Shiloh. **A Distributed Load Balancing Policy for a Multicomputer.** *Software—Practice and Experience*, Vol 15, No. 9, pp. 901–913, September 1985.

[Bayer79] R. Bayer, R.M. Graham and G. Seegmüller (editors). **Operating Systems - An Advanced Course.** Springer-Verlag 1979.

[Bersh87] B.N. Bershad, D.T. Ching, E.D. Lazowska, J. Sanislo and M. Schwartz. **A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems.** *IEEE Transactions on Software Engineering*, Vol SE-13, No. 8, pp. 880–894, August 1987.

[ANSA 89] **ANSA : An Engineer's Introduction to the Architecture.** *Architecture Projects Management Limited*, Release TR.03.02, November 1989.

[Birma85] K.P. Birman, T.A. Joseph, T. Raeuchle and A.E. Abbadi. **Implementing Fault-Tolerant Distributed Objects.** *IEEE Transactions on Software Engineering* Vol SE-11, pp. 502–508, June 1985.

[Birre84] Andrew D. Birrell and Bruce Jay Nelson. **Implementing Remote Procedure Calls.** *ACM Transactions on Computer Systems*, Vol 2, No. 1, pp. 39–59, February 1984.

[Black85] Andrew P. Black. **The Eden Programming Language.** *University of Washington Department of Computer Science*, Technical Report 85-09-01, September 1985.

[Black86] Andrew Black, Norman Hutchinson, Eric Jul and Henry Levy. **Object Structure in the Emerald System.** *University of Washington Department of Computer Science*, Technical Report 86-04-03, June 1986.

[Casav88] Thomas L. Casavant and Jon G. Kuhl. **A Taxonomy of Scheduling in General Purpose Distributed Computing Systems.** *IEEE Transactions on Software Engineering*, Vol 14, No. 2, pp. 141–153, February 1988.

[Chou 82] T.C.K. Chou and J.A. Abraham. **Load Balancing in Distributed Systems.** *IEEE Transactions on Software Engineering*, Vol SE-8, No. 4, pp. 401–412, July 1982.

[Cocks84] W.P. Cockshot, M.P. Atkinson, K.J. Chisholm, P.J. Bailey and R. Morrison. **Persistent Object Management System.** *Software—Practice and Experience*, Vol 14, pp. 49–71, 1984.

[Coulo88] George F. Coulouris and Jean Dollimore. **Distributed systems: Concepts and Designs.** Addison-Wesley, 1988.

[Cox 87] Brad J. Cox. **Object Oriented Programming : An Evolutionary Approach**. Addison-Wesley, 1987.

[Dasgu86] Partha Dasgupta. **A Probe Based Monitoring Scheme for an Object-Oriented Distributed Operating System.** *OOPSLA 86 Proceedings, ACM SIGPLAN Notices*, Vol. 21, No. 11, pp. 57–66, September 1986.

[DIX 80] Digital Equipment Corporation, Intel Corporation and Xerox Corporation. **The Ethernet Specification.** September 1980. Reproduced in *ACM SIGCOMM Computer Communications Review*, Vol. 11, No. 3, July 1981.

[Dixon88] G.N. Dixon. **Object Management for Persistence and Recoverability.** PhD Thesis *University of Newcastle upon Tyne, Computing Laboratory*, TR-276, December 1988.

[Eager85] D.L. Eager, E.D. Lazowska and J. Zahorjan. **A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing.** *Performance Evaluation*, Vol 6, pp. 53–68, 1986.

[Eager86] D.L. Eager, E.D. Lazowska and J. Zahorjan. **Adaptive Load Sharing in Homogeneous Distributed Systems.** *IEE Transactions on Software Engineering*, Vol SE-12, No. 5, pp. 662–675, May 1986.

[Elsho88] I.J.P. Elshoff. **A Distributed Debugger for Amoeba.** *Centre for Mathematics and Computer Science (CWI), Amsterdam*, Report CS-R8828, July 1988.

[Enslo78] Philip H. Enslow, Jr. **What is a "Distributed" Data Processing System?** *Computer*, pp. 13–21, January 1978.

[Garci84] H. Garcia-Molina, F. Germano Jr. and W.H. Kohler. **Debugging a Distributed Computing System.** *IEEE Transactions on Software Engineering*, Vol SE-10, No. 2, pp. 210–219, March 1984.

[Goldb83] A. Goldberg and D. Robson. **Smalltalk-80: The Language and its Implementation.** Addison-Wesley, 1983.

[Goldb84] A. Goldberg. **Smalltalk-80, The Interactive Programming Environment.** Addison-Wesley, 1984.

[Griet89] J.J. van Griethuysen. **Open Distributed Processing (ODP).** Invited paper at *IFIP 6.1 Conference on Protocol Specification, Verification and Testing*, June 1989.

[Gurwi86] Robert F. Gurwitz, Michael A. Dean and Richard E. Schantz. **Programming Support in the Cronus Distributed Operating System.** *Proceedings IEEE 6th International Conference on Distributed Computing Systems*, pp. 486–493, May 1986.

[Hać 86] Anna Hać and Theodore J. Johnson. **A Study of Dynamic Load Balancing in a Distributed System. Proceedings ACM SIGCOMM '86 Symposium**, pp. 348–356, August 1986.

[Halbe87] Daniel C. Halbert and Patrick D. O'Brien. **Using Types and Inheritance in Object-Oriented Languages.** *Proceedings of European Conference on Object Oriented Programming*, pp. 20–31, June 1987.

[Herli82] M. Herlihy and B. Liskov. **A Value Transmission Method for Abstract Data Types.** *ACM Transactions on Programming Languages*, Vol 4, No. 4, pp. 527–551, April 1982.

[Herli86] M. Herlihy. **A Quorum-Consensus Replication Method for Abstract Data Types.** *ACM Transactions on Computer Systems*, Vol 4, No. 1, pp. 32–53, February 1986.

[Hopp86a] Andrew Hopper, Steven Temple and Robin Williamson. **Local Area Network Design.** Addison-Wesley, 1986.

[Hopp86b] Andy Hopper and Roger M. Needham. **The Cambridge Fast Ring Networking System (CFR).** *University of Cambridge Computer Laboratory*, Technical Report No. 90, June 1986.

[Hsu 86] Chi-Yin Huang Hsu and Jane W.-S. Liu. **Dynamic Load Balancing Algorithms in Homogeneous Distributed Systems.** *Proceedings IEEE* 6^{th} *Internantional Conference on Distributed Computing Systems*, pp. 216–223, May 1986.

[Hwang85] Kai Hwang and Faye A. Briggs. **Computer Architecture and Parallel Processing.** McGraw-Hill, 1985.

[ISO 81] **Data Processing — Open Systems Interconnection — Basic Reference Model.** ISO 7489, 1983. Reproduced in *ACM SIGCOMM Computer Communications Review*, Vol 11, No. 2, pp. 15–65, April 1981.

[ISO 88] ISO/IEC JTC1/SC21/WG7 **Modelling Techniques for the Specification of the ODP Reference Model.** Document N 022, June 1988.

[Jones79] Anita K. Jones. **The Object Model: A Conceptual Tool for Structuring Software.** In [Bayer79], pp. 8–16.

[Jones86] Michael B. Jones and Richard F. Rashid. **Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems.** *OOPSLA '86 Proceedings, ACM SIGPLAN Notices*, Vol 21, No. 11, pp. 67–77, September 1986.

[Joyce87] J. Joyce, G. Lomow, K. Slind and B. Unger. **Monitoring Distributed Systems.** *ACM Transactions on Computer Systems*, Vol 5, No. 2, pp. 121–150, May 1987.

[Jul 87] Eric Jul, Henry Levy, Norman Hutchinson and Andrew Black. **Fine-Grained Mobility in the Emerald System.** *University of Washington Department of Computer Science*, Technical Report 87-02-03, February 1987.

[Jul 88] Eric Jul. **Object Mobility in a Distributed Object-Oriented System.** PhD. Thesis, *University of Washington*, TR 88-12-06, December 1988.

[Karge88] Paul Karger. **Improving Security and Performance for Capability Systems.** PhD. Thesis, *University of Cambridge, Computer Laboratory*, TR-149, October 1988.

[Lamps81] B.W. Lampson, M. Paul and H.J. Siegert (editors). **Distributed Systems — Architecture and Implementation.** Springer-Verlag (Second Edition 1983).

[Lazow81] Edward D. Lazowska, Henry M. Levy, Guy T. Almes, Michael J. Fischer, Robert J. Fowler and Stephen C. Vestal. **The Architecture of the Eden System.** *ACM SIGOPS Conference Proceedings 1981*, pp. 148–159.

[LeLan81] Gerard LeLann. **Motivations, objectives and characterization of distributed systems.** In [Lamps81], pp. 1–9.

[Lesli84] I.M. Leslie, R.M. Needham, J.W. Burren and G.C. Adams. **The Architecture of the Universe Network.** *ACM SIGCOMM Computer Communications Review*, Vol 14, No. 2, pp. 2–9, June 1984.

[Low 88] Colin Low. **A Shared, Persistent Object Store.** *Queen Mary College, Department of Computer Science*, Technical Report 450, February 1988.

[MacDo87] M.H. MacDougall. **Simulating Computer Systems, Techniques and Tools.** The MIT Press, 1987.

[Maeka87] M. Maekawa, A. Oldehoeft and R. Oldehoeft. **Operating Systems, Advanced Concepts.** Benjamin/Commings, 1987.

[Makpa88] Mesaac Makpangou and Marc Shapiro. **The SOS Object-Oriented Communication Service.** *INRIA Research Report*, No. 801, March 1988.

[Methf87] R. Methfessel. **Implementing an Access and Object Oriented Paradigm in a Language that Supports Neither.** *ACM SIGPLAN Notices*, Vol. 22, No. 4, pp. 83–92, April 1987.

[Mitra82] I. Mitrani. **Simulation Techniques for Discrete Event Systems.** Cambridge University Press, 1982.

[Mirch86] Ravi Mirchandaney and John A. Stankovic. **Using Stochastic Learning Automata for Job Scheduling in Distributed Processing Systems.** *Journal of Parallel and Distributed Computing*, Vol 3, pp. 527–552, 1986.

[Mizun87] Masaaki Mizuno and Arthur E. Oldehoft. **An Access Control Language for Object-Oriented Programming Systems.** *Kansas State University, Department of Computing and Information Sciences*, Technical Report CS-76-12, November 1987.

[Mulle87] Sape J. Mullender. **Process Management in a Distributed Operating System.** *Centre for Mathematics and Computer Science (CWI), Amsterdam*, Report CS-R8713, March 1987.

[Neuma89] B. Clifford Neuman. **The Virtual System Model for Large Distributed Operating Systems.** *University of Washington*, Technical Report 89-01-07, April 1989.

[Ni 85] Lionel M. Ni, Chong-Wei Xu and Thomas B. Gendreau. **A Distributed Drafting Algorithm for Load Balancing.** *IEEE Transactions on Software Engineering*, Vol SE-11, No. 10, pp. 1153–1161, October 1985.

[Parri88] G.D. Parrington. *Management of Concurrency in a Reliable Object Oriented Computing System.* PhD. Thesis *University of Newcastle upon Tyne, Computing Laboratory*, TR-277, December 1988.

[Pasco86] G. Pascoe. **Encapsulators: A New Software Paradigm in Smalltalk-80.** *OOPSLA'86 Proceedings, ACM SIGPLAN Notices*, Vol. 21, No. 11, pp. 341–346, September 1986.

[Patel88] Manjula Patel. **The Joint Academic Network JANET.** *University of Manchester Department of Computer Science*, Technical Report UMCS-88-6-2, June 1988.

[Quart86] John S. Quarterman and Josiah C. Hoskins. **Notable Computer Networks.** *Communications of the ACM*, Vol 29, No. 10, pp. 932–971, October 1986.

[Renes88] Robbert van Renesse, Hans van Staveren and Andrew S. Tanenbaum. **Performance of the World's Fastest Distributed Operating System.** *ACM SIGOPS Operating Systems Review*, Vol 22, No. 4, pp. 25–34, October 1988.

[Rents82] Tim Rentsch. **Object Oriented Programming.** *ACM SIGPLAN Notices*, Vol 17, No. 9, pp. 51–57, September 1982.

[Schan86] Richard E. Schantz, Robert H. Thomas and Girome Bono. **The Architecture of the Cronus Distributed Operating System.** *Proceedings IEEE 6th International Conference on Distributed Computing Systems*, pp. 250–259, May 1986.

[Schan87] Richard Schantz, Ken Schroder and Paul Neves. **Resource Management in the Cronus Distributed Operating System (Extended Abstract).** *ACM SIGCOMM Computer Communications Review*, Vol 17, No. 5, pp. 243–244, August 1987.

[Shapi86] Marc Shapiro. **Structure and Encapsulation in Distributed Systems : The Proxy Principle.** *Proceedings IEEE 6th International Conference on Distributed Computing Systems*, pp. 198–204, May 1986.

[Shriv88] S.K. Shrivastava, G.D. Dixon, F. Hedayati, G.D. Parrington and S.M. Wheater. **A Technical Overview of Arjuna: A System for Reliable Distributed Computing.** *University of Newcastle upon Tyne, Computing Laboratory*, TR-262, July 1988.

[Smith85] Edward T. Smith. **A Debugger for Message-Based Processes.** *Software—Practice and Experience*, Vol 15, No. 11, pp. 1073–1086, November 1985.

[Smith88] Jonathan M. Smith. **A Survey of Process Migration Mechanisms.** *ACM SIGOPS Operating Systems Review*, Vol 22, No. 3, pp. 28–40, July 1988.

[Snyde86] Alan Snyder. **Encapsulation and Inheritance in Object-Oriented Programming Languages.** *ACM OOPSLA '86 Proceedings*, pp. 38–45, September 1986.

[Stone77] Harold S. Stone. **Multiprocessor Scheduling with the Aid of Network Flow Algorithms.** *IEEE Transactions on Software Engineering*, Vol SE-3, No. 1, pp. 85–93, January 1977.

[Strou88] Bjarne Stroustrup. **What is "Object-Oriented Programming"?** *IEEE Software*, pp. 10–20, May 1988.

[Tane81a] Andrew S. Tanenbaum. **Computer Networks.** Prentice/Hall International, 1981.

[Tane81b] Andrew S. Tanenbaum and Sape J. Mullender. **An Overview of the Amoeba Distributed Operating System.** *ACM SIGOPS Operating Systems Review*, Vol 15, No. 3, pp. 51–64, July 1981.

[Tanen85] Andrew S. Tanenbaum, Sape J. Mullender and Robbert van Renesse. **Distributed Operating Systems.** *Vrije Universiteit Amsterdam*, Report IR-104, July 1985.

[Tanen86] Andrew S. Tanenbaum, Sape J. Mullender and Robbert van Renesse. **Using Sparse Capabilities in a Distributed Operating**

System. *Proceedings IEEE 6th International Conference on Distributed Computing Systems*, pp. 558–563, May 1986.

[Theim86] Marvin M. Theimer. **Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems.** PhD. Thesis. *Stanford University Department of Computer Science*, Technical Report STAN-CS-86-1128 (also numbered CSL-86-302), June 1986.

[Varad88] R. Varadarajan and E. Ma. **An Approximate Load Balancing Model with Resource Migration in Distributed Systems.** *Proceedings 1988 International Conference on Parallel Processing*, pp. 13–17, August 1988.

[Wang 85] Yung-Terng Wang and R.J.T. Morris. **Load Sharing in Distributed Systems.** *IEEE Transactions on Computers*, Vol C-34, No. 3, pp. 204–217, March 1985.

[Watso81] Richard W. Watson. **Distributed, System Architecture Model.** In [Lamps81], pp. 10–43.