VRIJE UNIVERSITEIT TE AMSTERDAM

PRINCIPLES OF DISTRIBUTED OPERATING SYSTEM DESIGN

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor in
de wiskunde en natuurwetenschappen aan
de Vrije Universiteit te Amsterdam,
op gezag van de rector magnificus
dr. P. J. D. Drenth,
hoogleraar in de faculteit der sociale wetenschappen,
in het openbaar te verdedigen
op donderdag 31 oktober 1985 te 13.30 uur
in het hoofdgebouw der universiteit, De Boelelaan 1105

door

SAPE JURRIËN MULLENDER

geboren te Nieuwer Amstel

1985

MATHEMATISCH CENTRUM, AMSTERDAM

Promotor : prof. dr. A. S. Tanenbaum
Referent : dr. P. J. Weinberger

CONTENTS

PREFACE

Distributed operating system research is in interesting combination of theory and practice. Problems are encountered in practice; theory is the tool for providing solutions. These solutions are used in system design and implementation. Practise needs theory, but theory needs practice too: Practice provides the inputs for theory and the feedback for the usability of solutions.

This thesis reports mostly about theoretical work on distributed systems, but this theoretical work had to be inspired by practical work, the ***Amoeba*** Distributed Operating Systems Project. Designing and building ***Amoeba*** uncovered many new research problems and provided a testbed for research results.

The Amoeba Project has been the project of a team, and if it weren't for the members of the team, much of this work could not have been done. I would like to thank the students who participated in the project by implementing various parts: Dick Biekart and Bram Janssen, who built the Block Server and the first of our three File Servers; they discovered that non-blocking transaction calls are far from ideal and invented a simple *task* mechanism. Jos Gutter and Jacquelien Jorissen implemented the transaction mechanisms and helped us discover many flaws in the initial design; these have been taken into consideration in the current design. Theo van der Meer and Carl Welman made a Directory Server which allows storage of capabilities in encrypted form. Theo van der Storm and Sjoerd Mullender have been working on links between UNIX and ***Amoeba***; Theo made UNIX files accessible to ***Amoeba*** software, and Sjoerd made UNIX software run on ***Amoeba***.

A few staff members also participated, both at the Vrije Universiteit and at CWI: Leo van Moergestel made the hardware work, Dick Grune made a pilot implementation of the Amoeba File Server, and Erik van Doorn helped me with the mathematics of *macro runs*; the analysis in § 5.4.1. is due to him.

Paul Vitányi and I did the work on locating ports together. Section 2.6 has been adapted from our paper in the 1985 PODC conference. Paul also gave some valuable comments on the presentation of this thesis.

It was fortunate that David Chaum was a guest at the Vrije Universiteit when I was working on the protection mechanisms of Chapters 2 and 3. The ideas presented there crystallised in many discussions we had on the subject.

Evert Wattel combines great skill in the application of mathematical tools with an excellent understanding of practical problems in system design. Together, we found the *Lighthouse Locate* algorithm of § 2.6 and we developed the mathematical models used in Chapter 5.

I would like to thank Peter Weinberger for the work he did as my *referent* and for being my host in New Jersey when we discussed his comments on my thesis.

Robbert van Renesse's contribution to the ***Amoeba*** project cannot be overestimated. Robbert implemented the transaction protocol and made it faster than any protocol we know. He redesigned and built the Amoeba Kernel which now has evolved beyond what is described in this thesis. He suggested many improvements to nearly every part of the Amoeba system. I am very grateful to him; the Amoeba project would not have been what it is without him.

Andy Tanenbaum must have suffered terribly under my Ph.D. research. During the first years of my work I took any excuse to program rather than study and he often had to prise me loose from the terminal and drag me back to work. Fortunately for both of us, Andy gradually managed to teach me that doing research and writing papers is fun. He is an excellent writer himself, and, when I offered my papers to him for inspection, he would return them covered with corrections, suggestions for improvements, and general comments on writing style. I managed to maintain a personal touch, however, by using *'spell -b'** on my thesis.

It was Andy Tanenbaum who put me on the trail of *'capability-based communication'* and this has proved a very fruitful research topic. It is a little-investigated area, but a very promising one: It provides a protection mechanism that is simple, easy to understand and requires a minimum of protected mechanisms.

Andy has been an excellent, though very demanding thesis advisor and I consider myself fortunate to have worked under him.

* Which was done by an American ...

1

INTRODUCTION

Distributed information processing has long been practised by living organisms. The human brain, one of the most complicated living organs, functions in a highly distributed manner; different parts of the brain have specialised in different functions, such as speech and sight. Yet there does not seem to be any central control in the brain, 'consciousness' cannot be pinpointed to one specific group of brain cells. However, although the human brain is the most sophisticated parallel processor known, it is not the only living organism functioning under distributed control. Most life forms use distributed control of some form or another. Even simple life forms, such as the one-celled *amoeba*, which have no single 'command centre' to decide where to go and how to get there, are somehow capable of co-ordinated action.

Imitating nature in all aspects, man has finally begun to incorporate the principles of distributed information processing in his most complicated artifacts, computers. In their desire to construct better, faster and more reliable information processing systems, researchers are building networks of many computers, co-operating to do their task more quickly and more reliably. The time is right for distributed systems research. The price of a powerful microprocessor chip will soon be a few dollars. Only a few years ago computers were used by a privileged few; today computers are invading our homes and offices to be used for a thousand and one different purposes. As computers become more popular, the need will grow to share the information and knowledge stored in them. It will become necessary to allow computers to exchange programs, documents, to exchange knowledge, in short, it has become necessary to connect computers in computer networks.

The technology for connecting computers is available; many varieties of local-area networks are on the market, and most are fast and reliable. However, the infrastructure which is necessary to manage and control distributed information has hardly been developed. The subject of this thesis is the design of such an infrastructure, a model that allows people to understand distributed computer systems and describe their actions.

Distributed computing is a new research area, one that introduces a whole range of new problems to be solved, problems of managing information systems without global and up-to-date information of their state, of finding ways to prevent inconsistencies in large bodies of data caused by unsynchronised simultaneous changes. Mechanisms for protecting information against unauthorised access must be found. The potential in distributed systems of much greater reliability must be used by designing services that can survive crashes of individual components of the system. For some of these problems, solutions had already been found in traditional, centralised operating systems; other problems did not even exist before the advent of distributed computing. Here these problems and others are examined, and some solutions are proposed.

This thesis discusses the issues in the design of a distributed information processing system. The realisation of distributed control in all parts of the system has been a key goal of the research; any centralised part would be a potential bottleneck when the system grows, and a liability in the face of crashes. It is because of the importance of distributed control that we have named the distributed operating system emerging from our research ***Amoeba***, after that one-celled creature using distributed control to move about.

Distributed control plays a central role in 'avoiding single points of failure.' Specialisation and control cannot be obtained through a simple hierarchical structure as exemplified by most armies. Again, the analogy with nature teaches us that extensive hierarchical systems can exist with a control structure that provides enough redundancy to survive 'simple' failures.

The material presented here is organised as follows. The remainder of Chapter 1 provides the background for this research, and gives an overview of related work. Interprocess communication is discussed in Chapter 2. In Chapter 3, the service model is introduced. This model provides an infrastructure for accessing objects, and for protection and accounting. Chapter 4 is about the operating system kernel, present in each host. In Chapter 5 principles are discussed and an algorithm is developed for scheduling processes in a distributed environment. A distributed file server is discussed in Chapter 6, and a service to assist users with accounting is the subject of Chapter 7.

1.1. Current Trends in Computer Systems

A number of developments in computer science research and in society have caused distributed systems research to become increasingly important. In this section we shall go into some of these developments and discuss the potential advantages of distributed systems over traditional systems.

1.1.1. Developments in Hardware

The main cause of the impact of computers on society in the past decade has been the dramatic drop in cost of computer hardware, combined with an enormous increase in capacity. Today, simple home computers are more powerful than the largest computers of 30 years ago. Although most microcomputers do not have the capacity of larger mainframes, they are so cheap, compared to large computers, that for many applications it is cheaper to use a number of microcomputers than one mini or mainframe. Although the cost of electronic components will steadily continue to drop, it must be noted that computer systems are not solely built of chips: terminals, printers, magnetic storage units, etc. will remain relatively expensive.

Traditionally, those devices that were connected to the computer were called **peripherals**, a once appropriate name, since the central computer was the all important machine, which was merely assisted by peripherals to obtain its input and produce its output. Although this view of computer systems is still firmly rooted in the minds of many computer centre managers, there are grounds to claim exactly the opposite nowadays. Peripherals are becoming far more expensive than the computers that control them; this is demonstrated already in some distributed systems, where sometimes computers are dedicated to controlling a single peripheral, just to make it run more efficiently.

For many peripherals (we shall continue to name them thus for want of a better term) there still is a strong effect of economy of scale: One big disk drive is cheaper than two small ones. Expensive peripherals, when used privately, are often under-used. This will induce users to want to share expensive peripherals, such as storage devices, laser printers, phototypesetters, graphic devices, etc. It is for this reason, that computer networks will ultimately make computer systems cheaper because many devices need not be replicated, but can be shared. Already these developments have caused local-area networks to become widespread in the last decade. Local networks make it possible to connect together large numbers of micros and minis and allow sharing of peripherals, and, above all, allow information to be shared between users of different computers.

An interesting consequence of the introduction of cheap computers has been a change in people's attitude towards operating systems. In the dim prehistoric days of computer science, operating systems as we know them today hardly existed; people used to sign up for an hour or so of computer time and had the whole machine to themselves during that period. Naturally, the computer spent most of its time waiting for the human programmer to feed it the necessary information. Computers were expensive machines in those days, and had to be used as efficiently as possible. Consequently, the first batch operating systems were soon developed, making it possible to keep the computer much better occupied: printing the output of program number one could coincide with running program number two, while program number three was already being read in. Operating systems grew more and more sophisticated; as computer memories grew larger, and many programs did not need all the available memory to run, more and more operations were allowed to overlap so the CPU could be used to utmost advantage.

When computers became somewhat cheaper, some computer capacity could be traded for more comfort to the programmer. Time sharing became increasingly popular in the seventies. So popular, in fact, that few, if any, computer scientists today will put up with batch systems any more. Time-sharing operating systems, however, are amongst the most difficult programs to write, debug and maintain. It is especially hard to give a maximum number of people reasonable service on a single computer. Vast amounts of money and effort have been spent on building time-sharing systems that support as many users as possible.

Now, this is also changing. Computers have become so cheap that, in most cases, it is no longer worth the effort to get the last grain of performance out of a computer. Today, when the capacity of a computer system becomes a problem, one simply buys another one. Distributed systems research has already produced some of the technology that allows computer systems to co-operate and share files and devices. This development will continue to the point where all users have several CPUs at their disposal. We shall again be able to afford to let a single program sit all alone in a computer, doing nothing most of the time. The effort of multiprogramming, the risk of crashes of the operating system, of bugs causing security leaks, and, in some cases, the absence of hardware that supports time sharing, will make it far more attractive to install more computers.

Right now, distributed systems development is usually confined to local-area networks, but a similar development is starting on a wider scale. Wide-area networks will allow sharing of information, software and special purpose hardware in a manner similar to local-area networks. Users can do most of their work on local machines, but occasionally gain access to, for instance, centralised number crunchers for solving special problems. At the moment virtually no support exists for efficiently exploiting the possibilities of wide-area networks.

1.1.2. Office Automation

The computer manufacturers have become increasingly interested in office automation. If their advertisements can be believed, an office is hardly worth being in if it is not equipped with personal work stations, word processors and what-have-you. Although it is an exaggeration to say that no office should be without a secretarial workstation, it is probably fair to say that office automation will become an important branch of computer industry, and hence undoubtedly of some interest to computer science.

A typical environment for office automation systems will be formed by a cable, running through the office building, with various workstations attached. Most workstations will consist of a cpu, a megabyte or so of memory, a keyboard and a high-resolution bit-map display. There will also be specialised machines:

file servers, printer servers, gateways to other networks, etc.

In such an environment, it is not possible to rely on the individual operating systems in each of the workstations for protection: it is too easy to replace the operating system, or even the whole computer, by one without the proper protection mechanisms. This issue has been an important consideration in the design of the interprocess communication mechanisms of Chapter 2.

The activities in the office environment described above will consist of document preparation, electronic mail and various forms of filing activities. Information sharing is an important aspect of office systems, so storage facilities must be provided that allow sharing of changing data. This requires data base techniques for maintaining the integrity of concurrently accessed, changing information.

1.1.3. On Line Database Systems

Data base systems form an increasingly important aspect of our computerised society. Apart from the social and political aspects of maintaining large data bases, there are many unsolved technical problems concerning reliability, integrity and accessibility of large data bases.

Distributed data bases also become more important. Airlines, banks, and multinational companies are examples of companies that maintain distributed data bases. An airline reservation system can be used to illustrate the problem of maintaining a large distributed data base: it is possible to buy a seat on a KLM flight from New York to Amsterdam almost anywhere in the world. Mechanisms are needed to prevent simultaneously selling a seat to different people in different parts of the world, because copies of the data base may not be properly synchronised.

Distributed data base research has provided solutions to some of these problems, but has also raised many new ones. A distributed data base is potentially very reliable, because of the possibility to replicate the information on more than one site. Its accessibility can also be much higher than that of a centralised data base, because it is unlikely that all sites in a distributed system will be down simultaneously. But keeping all replicas of the data in a distributed system consistent is a problem. Efficiently updating information in a shared, distributed data base is also a problem. Both these problems receive considerable attention in current distributed data base research.

1.1.4. Specialisation

As computers are being used for more purposes in different kinds of applications, more specialised systems are developed: Huge, super-fast, number crunchers are used for weather predictions and for solving the numerical problems of theoretical physicists and chemists; tiny microcomputers are built into point-of-sale terminals to register what goods leave the shop so it can be quickly restocked;

specialised computers are being built to efficiently resolve data base queries; others are being designed to do highly parallel searches in large storage systems; specialised computers control robots and other fanciful devices.

In a distributed system, the advantages of specialised systems and general purpose systems can be combined. By itself, a super computer can not be used very effectively: it is an inefficient tool for things like program development and editing. But when combined with a relatively inexpensive system for these things, a super computer can be used far more cost-effectively. Distributed systems can provide an infrastructure for using specialised hardware for speed, efficiency and simplicity in using specialised systems for specialised problems in a general purpose environment.

1.1.5. Reliability

In centralised computer systems, a crash of the operating system, a power failure, or a hardware glitch in CPU or memory will bring the whole system down. Distributed systems have the potential that a local failure does not affect the system as a whole. This possibility receives much attention in distributed systems research. Designers of distributed operating systems make an effort to design the pieces that make up the system to be crash resistant; that is, to confine the effect of crashes as much as possible.

The components of a distributed system have to be made *fault tolerant*, that is, mechanisms must be built in to recover from the consequences of failures in hardware and software. Many different kinds of failure exist: transmitted packets may lose some bits due to noise on the cables, disk drives may stop working, rendering the information stored in them inaccessible for a period of time, user programs (or even whole operating systems) may crash, possibly leaving files, databases and other programs in some half-finished state.

When designing distributed services, it is necessary to keep fault tolerance in mind at all times: databases must be updated in such a way that a crash halfway through an update does not have disastrous effects on its contents, interprocess communication must be arranged in such a way that garbled information is detected and retransmitted, when using distributed services, the possibility that a crash destroys the expected result must be taken into account. Much of this thesis is concerned with these problems.

1.2. Issues in Distributed Computing

Some of the potential advantages of distributed systems over conventional, centralised systems have already been mentioned in the previous section. The few years of distributed systems research have led to some general principles that designers of distributed systems should follow. Some of these principles are vague, and it is not always clear how they should be realised. This section goes into the important principles governing distributed systems design.

1.2.1. Fault Tolerance

A potential advantage of a distributed system over a centralised system is that it can be made **fault tolerant**. Failure of a hardware component or a piece of software need not bring the system down; in many cases it need not even cause any processing to fail. Fault tolerant systems design is a fascinating area of computer science which is only just starting to yield its first results.

Fault tolerance in case of hardware failures can only be achieved by replicating hardware. Obviously, a single processor system cannot continue to operate if that single processor breaks down. In a multi processor environment, other processors can take over when one of them breaks down.

The same principle applies to data. If the only copy of an important file is lost—in a disk crash, for instance—it is lost for ever. If, however, files are replicated, that is, the information is present at several different sites, a single disk crash need not cause any files to be lost.

It must be clear that replication of data has consequences in terms of cost in storage and complexity of maintenance of multiple copy files. It is impossible to change two copies of a datum at exactly the same moment; there is a short time in which the copies of that datum differ, and other users could come to false conclusions by observing a datum during that short period of inconsistency.

An even greater problem with distributed data (plain files, or whole data bases), is that failures may occur in the middle of an operation on some data. Recovery from such failures is often difficult, because it is hard to tell in retrospect just where the failure occurred, which data were affected and which were not. For some types of processing recovery is simple. Take, for instance, a compiler that crashes half way through a compilation. In nearly all cases, the compiler can just be run again, and the only consequence of the crash is some delay in producing results. Recovery is simple, because a compilation does not really create new information; it just transforms existing information (the program to be compiled) into another form (a binary, or a list of errors). In many other cases, however, this type of recovery is not possible. The problem of recovering from crashes stands out in programs that operate on files or data bases, because, in contrast to the compiler example, an update on a datum does add new information to the system (and deletes old information).

Distributed storage systems or data base systems designers usually aim to achieve three goals:

- To the users, replication of data is transparent; that is, all copies of a replicated datum appear to change at exactly the same moment. The user is only aware of an utterly reliable file store.
- The crash recovery mechanisms in the storage system or data base system always restore the system to a *consistent* state after a crash. (This issue is discussed in detail in Chapter 6).
- The additional cost of maintaining multiple copies of data and crash recovery mechanisms must be small during normal operation. Although crash recovery

mechanisms are an essential part of distributed storage systems, crashes are the exception rather than the rule, so crash recovery may be an expensive operation, provided the normal day-to-day operation is only slightly less efficient because of anticipation of possible crashes.

Fault tolerance recurs at every level of the design of a distributed system. Every level must cope with the errors and failures that may occur at that level, and with the failures that the next lower level cannot cope with by itself. It is usually impossible that each layer of a system can recover from each possible error in that layer, so every layer must be prepared to do some error recovery for lower layers.

An aspect of distributed systems, closely related to fault tolerance, is that events cannot always be immediately observed everywhere in the system. When a computer crashes, it takes some time before this is noticed by the other computers of the system. But also, when one server process creates a new file, the other servers are not aware of this at the same moment.

In distributed systems, events are not observed immediately, not all events are always visible everywhere, and the order in which events are observed may be different in different locations. This can have serious effects on the view different processes have of the system, and can it can cause different results in different places or at different times that should have been the same. Dealing with this problem (often referred to as **signal observability**) is made easier in a fault tolerant environment: the effect of a decision made, based on failure to observe some event can often be limited or cancelled using error recovery techniques.

1.2.2. Availability

Replication of hardware and software can lead to higher availability; given a fault tolerant design, crashed processes or processors need not lead to discontinuation of any services. Although replication of services and data is essential to continuous availability in the face of hardware and software crashes, additional provisions have to be made in order to ensure uninterrupted service while parts of the service are down.

Even when replicated hardware and software is available it is not trivial to provide continued service when parts can crash. No aspect of a service may be controlled by a single server, because a crash of that server would render that aspect unavailable. It is thus necessary that every resource, be it a physical or a virtual one, be controlled by at least two controllers. This principle is referred to as **distributed control**.

Although it is not difficult to realise the necessity for distributed control, it is surprisingly hard to design and build services that use distributed control at all levels. Different forms of distributed control exist. At its simplest, distributed control can be realised by a form of centralised control, backed up by sleeping colleagues; the active controller periodically sends virtual *sleeping pills* to its colleagues to keep them passive. When it crashes the supply of sleeping pills stops, and one of the sleeping controllers wakes up and starts sending sleeping pills to the others; after some time, it can be sure the others are safely asleep, and it can take over control. This form of distributed control only works if no state information is lost in a crash. Many other forms of distributed control exist, based on *voting*,[20, 76] *token passing*,[38] *timestamps*,[54] *locking*,[21] etc.

To ensure the availability of a distributed system, it is usually not enough to provide replicated services; accessibility of the information is at least as important as accessibility of the services that operate on it. When information is stored on just one host, or just one disk, it becomes inaccessible when that host, or that disk crashes. In the case of a serious disk crash (*e.g.*, a head crash) the consequences are even worse, the information on that disk is lost forever.

The only safe way of storing information such that it is not lost when disks crash is to replicate the information and store it on different disks, preferably attached to different hosts as well. Although this increases the reliability of information storage, it does not automatically increase its availability: The algorithms that update replicated information may well require that all copies of the information can be accessed simultaneously. Such algorithms, in the worst case, could decrease the availability of stored information, because the probability of a single host and a single disk drive being up is far greater than the probability that all of a number of hosts and disk drives are up.

Algorithms for handling replicated information must be designed with great care. Replication serves two goals: increased reliability and increased availability. It is easy to meet just one of these goals, the mechanisms needed to meet both are the subject of file system and data base research all over the world.

What applies to information also applies to hardware: constant availability can only be guaranteed by replication. A reliable distributed system needs many processors, many disk drives, many terminals, printers, etc. However, the available hardware must also be controlled properly to be useful in offering better service to its users. A distributed system, made up of many personal computers, or many private workstations, does not appear more accessible to someone whose workstation just stopped working: the system does not automatically provide a replacement. At the same time, when all works well, the performance of such a system is not better than that of a single personal computer or workstation. Both performance and availability (and also reliability) can be increased by pooling resources: all workstations and computers are thought of as being in a pool where they are allocated to whoever needs them. When one machine crashes, it means that there is just one machine less in the pool; service is affected only marginally. Another consequence of pooling resources this way is that it lends itself better to the bursty nature of computing: Most users tend to need little computing power most of the time, and a lot of it occasionally. Dynamic allocation from a pool is efficient for such an environment. Although processors were taken as an example of the possibility to pool resources, the same advantages hold for printers, disk space, tape drives, and many services.

1.2.3. Modularity

The need for structured design of large software systems cannot be stressed enough;[11] in distributed systems design this principle is even more important. The interactions between parts of a distributed systems are far less predictable than in centralised systems. Parallel processing in distributed systems may cause the deterministic behaviour of some processes to be lost. Debugging a distributed application is much more difficult than debugging a centralised one. Distributed systems must be designed with even more thought to structure and modularity than conventional centralised systems.

Modular design has many advantages; it leads to better understanding of how the system works; hardware dependencies can be collected in few places only; failures are more easily understood, so recovery is simpler; modules that do one thing only can be re-used in other applications; software becomes easier to test and debug; the system becomes more flexible and it is easier to change algorithms, or adapt modules to changed circumstances.

As an example of how a modular design can lead to a system that is easier to understand, and simpler to modify and adapt to other environments or applications, consider a data base system, using a layered structure.* The system consists of four layers or modules: disk service, distributed storage service, data base service, and an application process that uses the system. The principle of modular design has made it possible to separate the issues of replicated storage, concurrency control, query processing and the application itself. Replicated storage is handled transparently by the disk servers, allowing designers of other modules to forget about the problems of simultaneously updating two copies of a disk block. Concurrency control is handled by the storage servers, so the designers of the data base system can concentrate on the representation of the information, on query processing, and on data base problems in general, without having to concern themselves with thinking about what may go wrong when two updates occur simultaneously. data base management is done by the data base service, so an application programmer need not think about how the information is represented; the application programmer can use the commands provided by the data base system, without having to know how these commands are implemented.

In spite of perfect modular design, the separation of functionality can never be totally transparent. It is not always possible for each module to recover from every error or failure; some failures must be reported to higher layers to be dealt with at another level. As an example, consider concurrency control: the file server normally resolves threatening concurrency conflicts, but it cannot recover from every conflict without knowledge of the *meaning* of the update. However, the *effect* of a failure can be minimised and tools can be provided to higher layers in the system for efficient and simple recovery from failures.

* The File Service, described in Chapter 6 provides such layering.

A central issue in the research presented in this thesis is the design of an infrastructure, the design of tools that facilitate and encourage modular design of distributed systems. The paradigms used to describe the interactions between modules are the **client**, **service** and **object**. A service is a module that provides a set of commands and responses to a client process for accessing and operating on objects. The service model can be compared to *remote procedure calls*,[68] and to *abstract data types*,[40] but it is not quite the same. The service model is described in detail in Chapter 3.

1.3. Current Research in Distributed Operating Systems

The volume of research on distributed systems indicates it is a popular subject. A large amount of this research concentrates around various aspects of distributed office systems and general purpose distributed systems, the area of this thesis.

The following sections will discuss work currently in progress in this area at research sites around the world.

1.3.1. Infrastructure

It is vital for the users of an operating system that they perceive an easy-to-understand interface to the system. The system must be built out of modules, each with a simple and straightforward task, that can be logically fitted together.

There is a trend towards *open systems*, systems with an operating system kernel that is kept as small as possible, most services being provided by user programs. Among of the first advocates of such systems were B. W. Lampson and R. F. Sproull, who made a plea for open systems in [36]. Open systems have many advantages over closed systems; an open system has a smaller kernel, so the kernel is often faster, can be debugged more easily, and becomes more portable. Most services, traditionally in the operating system, are now provided by (privileged) user programs. This makes these services more accessible to maintenance and makes it easier to install new versions, test variants while the system is running, and it allows several competing services to co-exist side by side.

Distributed Systems often use the *service model* in one form or another. This model exists in variations: *object model*,[29] *abstract data types*,[40] or *remote procedure call*.[68] The underlying principle is that each *object* is managed by a *server*. *Client* processes can only access an object by making a *request* for an operation on the object to its server. The server carries out the request and returns a *reply*. This model is related to abstract data types, because the server defines a set of commands and responses on its objects, much like an abstract data type defines a set of operations on data types. It is related to the remote procedure call mechanism, because clients send requests and await replies similar to the remote call and return in the remote procedure call mechanism.

One infrastructure for supporting this model has been discussed in [79], where R. W. Watson and J. G. Fletcher distinguish a four layer hierarchy. At the bottom layer an Interprocess Communication facility is provided. On top of that is a Service Support Layer which deals with protection, naming, authentication, error control, synchronisation, etc., issues that recur in most services. On top of the Service Support Layer is the Service Layer, with system-provided services, such as file service, process service, clock service, data base services, etc. The highest layer is the Customer Layer; at this layer client processes execute and make use of the services provided by lower layers.

A layered design encourages a modular structure and makes the system more understandable. The work presented in this thesis follows Watson and Fletcher's hierarchy, although, some aspects (*e.g.*, protection) have been moved to other levels.

The primitives ***Amoeba*** provides to the programmer for exchanging requests and replies loosely resemble those of *Thoth*[8] and *V*.[9] Where *Thoth* uses *.send*, *.receive* and *.reply*, ***Amoeba*** has *trans*, *getreq* and *putrep*. Here the resemblance stops, the mechanism for addressing services and objects, and the associated protection mechanism used by ***Amoeba*** is radically different.

1.3.2. Protection

Most distributed systems provide protection mechanisms to prevent unauthorised access to objects and resources. Conceptually, there is a matrix with a row for each object and a column for each user. At the intersection of an object's row and a user's column the *rights* can be found that the user has on the object. Each operation on an object is protected by right. A user who has that right may carry out the operation; a user who does not have the right is not allowed to carry out the operation.

In a real system this matrix is never represented as such; it is always cut up. It can be cut up in several ways. Some operating systems store the matrix row by row. Attached to each object would be the row of the matrix, corresponding to that object. When a user wished to access an object, the operating system would check the user's permission to carry out the requested operation before allowing the access. A row of the matrix is called an **access control list**. Access control lists take up much space, but they can be condensed at the cost of some generality: the UNIX* operating system for instance, uses three rights to indicate the allowed operations by the *owner* of an object, users in the owner's *group* and *others*, requiring nine bits for the access control list. Many other systems use similar protection structures.

The other method is to split up the matrix the other way; that is, the matrix is split up in columns, one per user. With this method, the operating system maintains a per-user list of objects and rights on the objects, called a **capability list**.[15]

* UNIX is a Trademark of AT&T Bell Laboratories.

A capability for an object can be compared to a ticket for a concert: just like a ticket gives the holder the right to attend the concert, a capability gives the holder the right to carry out the indicated operations on the object.

Distributed operating systems are more often capability systems than centralised systems,[81,78,53,67] although it is not clear whether this is *because* they are distributed systems.

Usually, capabilities are managed by the operating system to prevent forgery. If user processes could store capabilities themselves, it would be easy to modify them to include more rights, or to construct new capabilities for objects that the user has no right to access. However, it is possible to allow users to manage their own capabilities. This requires mechanisms to prevent users from forging their own capabilities, such as the mechanism discussed in [19]. This method uses public key encryption to protect capabilities. In Chapter 3 we shall propose another method for protecting capabilities in user space which does not require (expensive) public key encryption.

Protection mechanisms need authentication mechanisms, both when access control lists are used and when capabilities are used. In the former case authentication mechanisms are needed for the server, to determine the identity of users making requests, in order to find out from the access control lists whether access may be granted. In the latter case, authentication mechanisms are necessary for the user who does not like to see a capability passed to a server fall in the hands of an imposter.

In most operating systems, the operating system is responsible for providing authenticating information: all communication has to pass through the operating system kernel which checks permissions and provides authenticating information to the communicating peer process. Depending on the communication service, provided by the operating system, authenticating information is provided on a per-message basis or on opening a virtual circuit.

1.3.3. The Operating System Kernel

Most distributed systems have a copy of the distributed system kernel control each host. The kernel provides an environment for running processes and implements the interprocess communication and protection mechanisms. These kernels are usually based on the same design principles as traditional, centralised operating system kernels: they provide an execution environment for processes, and allow interprocess communication by means of *system calls*.

It is not clear whether this type of process environment is suitable for a distributed system. The traditional 'system call model' has some properties that may hinder the development of server programs. One problem with the traditional process environment is that system calls block. When a process makes a *read* system call, for example, the process blocks until the data is available. Blocking is not at all suitable for a server process that handles requests for many clients simultaneously. Such a process needs mechanisms that allow it to handle many requests simultaneously.

A problem of traditional process environments when implementing services is that it is not possible to express the multi programmed nature of most services in such an environment. Most service processes will handle requests from many users simultaneously in an event-driven manner: When the server comes up, it waits for its first request. When this comes in, the request is processed to the point the server must wait for i/o, or for the reply from a sub-server. The server is then free to handle new requests from other users. Eventually, an event will signal completion of whatever kept the server waiting on the first request, and the server can do some more processing.

This scenario requires a complicated administration of requests being processed, their state, and the pending event. Traditional operating systems provide no help in programming such servers; the only choice to be made is to design the server as a single-thread program which handles only one request at a time, or to design it as a complicated event-driven finite-state machine.

In Thoth[8] *teams* of processes can be formed, sharing an address space, to structure servers that can handle *n* requests simultaneously as *n* processes in a team. This eliminates the complication of handling many requests simultaneously in one process, without the inefficiency of many processes that cannot share caches, buffers, etc. The *clusters* of Chapter 4 are similar to teams, and provide a tool for the design of server programs.

1.3.4. Process Management

Apart from creating a process environment, a distributed system must also see to it that processes are actually run. When using services to model the interactions of processes, it seems natural to conceive a 'process service' that carries out requests to create and manipulate processes. Few distributed systems have implemented process service this way, but have implemented process management as an operating system function. From a historical viewpoint, this can be readily understood: the operating system is responsible for providing an execution environment for processes, so it must also be responsible for creating and running processes. Process management has many aspects, however, and it is not at all natural that every one of these aspects must be carried out by the operating system. The operating system should provide the mechanisms that allow process execution, but should not contain the policy decisions where processes should execute, or how different classes of processes should be scheduled.

Using the same operating system kernel, it should, for instance, be possible to implement both the 'Personal Computer Model,' (as exemplified by Xerox PARC[36]), and the 'Processor Bank Model,' (as used, for example, in the Cambridge System[50]). In the first model, every user has a personal computer that can be scheduled as the user sees fit; processes running in these personal computers can communicate with each other and make use of the services provided by the system. In the second model, all processors are pooled, and whenever a process is created (or a user logs in, as in the Cambridge system), a processor is allocated from the pool for a shorter or longer period. This requires a Process Service in the same vein as the Cambridge System's *Resource Manager*; the Process Service is outlined in Chapter 4.

Process scheduling is another important aspect of process management. There is extensive literature about scheduling, but, surprisingly, nearly all literature deals with the problems of *deterministic scheduling*.[26] In deterministic process scheduling, it is assumed that the execution times of the processes to be scheduled is known in advance. In a general purpose operating system this is virtually never the case. Still, the problem of *undeterministic scheduling* has received little attention in the literature, not to mention undeterministic scheduling in a distributed system. Scheduling is an important subject and still needs much research. Chapter 5 is a start in this direction.

1.3.5. Distributed File Systems

The open systems model has encouraged research in distributed file systems; it is much easier to design, implement and test a stand-alone file system, than one embedded in the operating system. Many file servers have emerged, ranging from simple ones, based on one server process and without concurrency control mechanisms, to very sophisticated ones, with many server processes which synchronise concurrent access to files to ensure the integrity of the file system.

Distributed file servers, implementing a concurrency control policy, are, of course, most interesting from a research viewpoint. Among these, it appears, *locking*[21] is the most widely used concurrency control mechanism.[71, 18, 24] The SWALLOW file server[55] is one of the few exceptions; it uses *timestamps* for concurrency control. Recently, however, *optimistic concurrency control*[33] has become a popular concurrency control mechanism in data base systems. Although optimistic concurrency control has attractive properties for a file server, it had not yet been used as the synchronisation mechanism in a distributed file system until the work reported in this thesis.

Few, if any, applications can live without the services of a file system. Distributed file systems must therefore be designed to be highly reliable and crash-resistant. Techniques for replicated storage of data are often used to guarantee access to files, even in the face of single-site crashes.[77, 45] Management of replicated files is a difficult problem, which has been addressed by researchers in data base systems.[20, 76, 77]

It is unfortunate that these problems have been addressed almost exclusively from the viewpoint of data base research. This has led to the development of large data base systems that comprise solutions for replicated data management, distributed concurrency control, and information management. A much clearer view of the problems and their solutions could be obtained if the issues of data base management, the issues of concurrency control, and the issues of replication management could be addressed separately. Although this will not necessarily lead to a system that is more efficient, it will almost certainly yield a system that is more flexible: A distributed file server with concurrency control is not suitable just for data base applications. It can be useful in a source code control system,[60] or in a text-processing environment.

Existing file servers have proven to be unsuitable as a basis for data base systems.[70,73] When designing a file server, it is therefore necessary to investigate the possibilities to separate a database system into a data base layer, which contains the management of *information*, and a file server layer, which manages reliable storage of the data needed by the data base layer. Results of this research can be found in Chapter 6.

1.4. Principles of Distributed Operating System Design

Surveying current research in general purpose distributed operating systems, we see some general principles and trends emerging. Principles incorporated in nearly every distributed system (in contrast to centralised systems) are to allow and encourage parallel processing, to accomplish a large degree of crash resistance and provide mechanisms for crash recovery. In realising these principles some general trends can be observed. Most distributed systems rely on the service model in one form or another; capability-based systems are becoming more popular; many distributed systems are open systems, that is, services are provided by processes running outside the operating system kernel; most distributed operating systems provide file service with some form of concurrency control and have a mechanism for atomic update. These trends have emerged through experience with various mechanisms as solid mechanisms for reliable fail-safe distributed operating systems. There is no reason not to use the service model, or protection with capabilities, or an open architecture.

However, this by no means implies that these issues form a closed research issue. Although mechanisms exist that implement capability based protection, file servers exist that use atomic update and provide concurrency control, protocols exist for client-server communication, etc., these systems are not perfect, and can still be much improved. In the following sections the background is given for carrying out just this research.

1.4.1. Protection

Many existing distributed operating systems rely on a distributed operating system kernel for protection. Usually, the kernel manages capabilities[81,78] for user processes, to prevent them from being forged. In nearly all distributed systems, the kernel handles authenticating information for user processes.

Reliance on the operating system kernel for protection has some serious disadvantages, however. In an environment with heterogeneous hosts, the operating system kernels must all be slightly different, since they all run on different hardware; the operating system interfaces may not be exactly the same. Yet, the kernels must all provide secure protection mechanisms. If the protection mechanisms of just one host can be circumvented, the security of the whole system is jeopardised.

Another, even greater, threat to security in a system that uses the kernel for protection, is posed by the impossibility to supervise all the hosts on the network. Distributed systems often consist of a cable snaking through an office buiding, with a socket in each room into which various hosts can be plugged. It is all too easy to take out the official 'approved' operating system and insert a special 'intruder' system that circumvents the built-in protection mechanisms. Usually it means replacing the operating system disk or floppy, sometimes it means replacing the operating system PROM or EPROM, or, in the simplest case, it means just booting another operating system binary.

Protection mechanisms that do not rely on the security of an operating system kernel have only been investigated as exercises in cryptography;[19] but have hardly been studied for the purpose of developing a practical protection mechanism in distributed operating systems. Chapters 2 and 3 are largely devoted to this problem.

1.4.2. Process Environment

In an open distributed system, the mechanisms for process interaction form an important aspect, because the amount of process interaction is much greater in an open system; file access requires communication with the file server process, terminal access requires communication with the terminal server, etc.

Processes, especially service processes, have different environmental requirements from processes in a centralised system: In a traditional system, processes make requests to the operating system via 'system calls,' comparable to subroutine calls to functions in the operating system kernel. The process blocks until the system call has completed. While this is perfectly satisfactory in centralised applications, and also for many distributed applications, it is a nuisance in many other distributed applications: If server processes would block on disk requests, they could not process any requests for other users during the time the disk is busy, nor could they have multiple outstanding disk requests, allowing the disk server to use an *elevator algorithm* to reduce disk arm movement.

A server process usually has to handle requests from many clients simultaneously. In a conventional programming environment, this requires an 'event driven' programming style: when an event occurs, the program looks up the associated client request and acts depending on the state of the request and the nature of the event.

The process environment, offered by a distributed operating system, must take these problems into account, and provide appropriate mechanisms to facilitate the construction of server processes. Most distributed systems to date have ignored this problem, yet we believe it is an important one. Chapter 4 describes an execution environment for processes functioning in a distributed system that is more versatile than traditional process environments. It allows both the traditional style of programming with a single thread of execution, and a new, distributed style of programming, with many execution threads.

Process management is also a poorly investigated area of distributed systems research. Often, each of the constituent operating system kernels making up a distributed system does its own process management; the distributed system is just a collection of operating systems whose processes co-operate. However, there are also distributed systems where process management is viewed as a distributed service. The available processors are pooled, and allocated to deserving processes. The Cambridge Model Distributed System[50] uses a processor pool, managed by the Resource Manager. Conceivably, processes can be migrated to other processors, for instance, to bring relief to a heavily loaded processor, or to bring a process closer to the resources it uses. Mechanisms for process migration have hardly been studied at all. These mechanisms are studied in Chapter 4, while scheduling algorithms for an environment where processes are allowed to migrate form the subject of Chapter 5.

1.4.3. Distributed File Service

Fault tolerance is one of the foremost goals of distributed systems design, so it is not surprising that research in distributed file systems has been a popular topic: magnetic storage devices usually survive crashes, so it is vital that their contents can be brought back to a consistent state after a crash. Another reason for associating fault tolerance with file system design is that most processing is on files; files are usually the source and sink of information processing. If the information on files can be kept consistent in the face of crashes, crash recovery becomes much simpler.

Among all the research being carried out in the area of distributed file systems, there are still some arid regions in the field, as indicated by two recent publications.[73, 70] It comes as a surprising realisation that file service as provided by many operating systems is too sophisticated. What causes file servers to be unusable for some applications is that they give too much service; especially database systems suffer from this problem.

A summing-up of the problems encountered in many file systems that render them unpractical for many applications is given in Chapter 6; a more elaborate summing up can be found in [73]. The **Amoeba File Service** was designed as a distributed file server that does not have these problems. Yet it does provide much of the functionality that other file servers offer, albeit in a slightly different form.

The file server, which is discussed in Chapter 6, uses *optimistic concurrency control,* a form of concurrency control that had previously only been used in database systems. Its version mechanism makes it suitable for a Source Code Control System[60] and it can be used on *write-once* media, such as the *optical disk*.

1.4.4. Accounting and Resource Control

Accounting has never been a popular research area. Perhaps, researchers are interested primarily in consuming resources and do not like being confronted with the bill. Most operating systems in use today provide some, often primitive, accounting and billing mechanisms. These do the accounting of what may be termed 'system resources,' such as CPU seconds, memory consumption, or disk space. The times are changing, however: Resources that were once expensive, such as processor time and memory consumption are now cheap, and resources that were never thought of as resources, such as compilers, text processing systems, etc.—resources that were considered extras that the computer centre's management provided to make computer use more attractive—are now the most expensive parts of the system. The cost of hardware has dropped at a steady rate in the past decades, due to better manufacturing processes, larger quantities and advances in VLSI technology. The cost of software has gone up, partly because of increased programmer wages, but mostly because of the increased complexity and size of most software systems used nowadays.

In traditional closed systems, all resources were provided by the 'computer centre,' so accounting was simply done by the computer centre, tailored to the computer centre's needs. In an open system, the computer centre is no longer the primary service provider; all users may start offering services to the rest of the user population.

These developments require a fresh look on accounting and resource control mechanisms. Accounting mechanisms are needed that allow each user that offers a service to have accounting done on the use of that service. Different users will want different accounting policies, so the accounting mechanisms must be general enough to allow these different policies to be realised.

Very little research has been done in this area, so the work presented in Chapter 7 is original. Only the work being done on the Osis[57] project is remotely related: it is an attempt to form an open market for offering and consuming services in a wide area network. In Osis bank services are available to arrange for payment of service consumption, much like the bank service of Chapter 7. However, the philosophy of authentication mechanisms underlying Osis is quite different from those in a distributed operating system.

2

INTERPROCESS COMMUNICATION

The Interprocess Communication Mechanism is a cornerstone of every distributed system and the design of such a mechanism has indeed been a key issue in our research. This has resulted in a capability-based, message-passing service. Capabilities are not just used as a protection mechanism between communicating processes, but find a much wider use to protect all objects, processes and services in the system.

This is an important concept: Objects and services can only be accessed through the interprocess communication mechanism, and the interprocess communication mechanism allows communication only with objects or services whose names are known. Names are capabilities in ***Amoeba***. An ordinary process, computing—say—prime numbers, can never be bothered with messages from a nasty neighbour process, unless it makes the name of its **port** public somehow. Other users can never access someone else's private files, unless the owner gives away the capabilities for them.

Putting the protection mechanism in the interprocess communication mechanism in the network has made it possible to allow any processor, with or without protection, with or without an operating system, under control of the network manager or of a malicious user, to be connected to ***Amoeba*** without posing a threat for the security of the system. This enables us to connect to ***Amoeba*** multiprogrammed minicomputers or mainframes with a secure operating system on the one hand, and cheap micros without memory management, protection hardware or a secure operating system on the other. An operating system can be simplified significantly because of the lesser demands made on it for protection and security; breaking the system's security becomes much harder because the protection mechanism is beyond the user's reach.

The interprocess communication mechanism is implemented as a hierarchy of layers, each layer augmenting the layers below it. The lowest layer, the physical layer, can be CSMA/CD, a ring, a mesh net, or almost any kind of other network.

On top of this layer is the **Port Layer**, which provides a simple datagram service, with an addressing scheme based on ports. This layer provides the system with its protection mechanism. Reliability comes in the **Transaction Layer**, the layer above the Port Layer. The Transaction Layer uses the unreliable (but secure) datagram facility of the Port Layer to offer reliable **transactions**. A transaction consists of a **request** from a client process to a server process, followed by a **reply** from the server to the client. User programs interface to the Transaction Layer. For emulation of existing operating systems, or further simplification of the user interface, an Emulation Layer can be put on top of the Transaction Layer.

This chapter describes the Port Layer in detail. The Transaction Layer is discussed in Chapter 3.

2.1. Naming, Addressing and Routing

The choice of a naming mechanism in a distributed system is an important one. Processes access services and objects through the network, and it is the naming mechanism that must see to it that messages addressed to a service, or to an object within a service, are delivered to the correct process.

A mapping is required to map the names of services and objects onto the names of service processes, and eventually onto an address of the service process. This mapping process is an essential aspect of any naming mechanism, and a key issue in the design of a naming mechanism is at what level this mapping must be done, where in the network it should be done, whether the mapping has to be done in several stages, and how protection of names can be ensured.

Names form a hierarchy identifying objects at different levels of the system, so it is possible that an object is referred to by different names at different levels of the system. The network designer's task is to choose simple, convenient and efficient names at each level, and, if possible, to reduce the number of levels to a practical minimum. At one end of the spectrum, the name of an object could be composed of the name of the processor where the object resides, the name of the process that manages the object, and a number used by the process to refer to the object. At the other end of the spectrum, the name of an object could be a character string, chosen by the owner, and interpreted in the owner's environment. The first example shows a name that can be conveniently used by an interprocess communication mechanism. The mapping function is trivial and delivery is easy. This scheme, however, has the disadvantage that such an object name is inconvenient for human users and that the name of the object changes as the object moves around the network, or as the manager process changes. The second example shows just the opposite; the name is convenient for human users: it can be independent of what service process currently maintains the object, and of the location of the object. However, mnemonic names can be inconvenient in several ways. Different users will choose the same name for their objects (*e.g.*, *'file'* for a file) and mechanisms have to be found to detect name clashes, or to give each user a different name space. Furthermore, a mnemonic name reveals nothing about the object's location, so mechanisms are also needed to find where an object is located when its name is presented.

In order to understand the issues, associated with naming, it is important to see the relation between a *name*, an *address*, and a *route*. A name indicates *what* one seeks, an address *where* it is, and a route *how* to get there.[5, 65] The name of a service or an object must be mapped onto a network address, and the address must then be mapped onto a route. In the most general case these mappings are separate mappings. When objects and services move through the network, their addresses change; the mapping of a name onto an address has to be adapted, but the mapping of addresses onto routes does not change. But when the topology of the network changes, or congestion occurs, the mapping of addresses onto routes may change, while the mapping of names onto addresses remains unchanged.

The choice of the levels of naming is an important one, and determines the usability and efficiency of a distributed system. In most distributed systems there is a hierarchy of naming levels. At the highest level we usually find location-independent, but user-dependent mnemonic names, convenient for human beings. At the lowest level we find location dependent, time dependent, machine dependent names, often totally unusable for humans. In between, a wide variety of naming methods exists.

It was a starting-point in the design of our naming scheme that there be one name space for all objects, services and processes, combined with a straightforward mechanism for mapping these names onto machine addresses. Techniques for mapping machine addresses onto routes are well known,[41, 42] and naturally they differ wildly for different network types. If possible, the mapping onto machine addresses and routes should be transparent to user processes, but without requiring an operating system in each host. Finally the algorithms that perform the mapping of names onto addresses must be distributed, avoiding the need for global knowledge in order to do the mapping.

2.1.1. Approaches to Naming and Addressing

A number of naming schemes are in use in existing distributed systems; some are suitable for the needs of systems such as ***Amoeba***, others are not. The most important level of naming is formed by the names provided at the interface between the system and the user programs; let us call these **system names**, for want of a better expression. Every user process must eventually map its internal names onto system names, and the system and network must map them onto host addresses and routes. A good choice is important for system naming; both the system and the users are affected by this choice, and changing over to another naming scheme is hardly feasible. User supplied subroutines, programs, or services can be used to convert more convenient or suitable higher level names onto system names, and vice versa.

We shall present four system naming mechanisms, as an illustration of the spectrum of possibilities. They are:

1. *A system name consists of a host name plus an object name on that host.* This rather primitive way of naming objects does not allow objects to move through the network without changing their name, but greatly simplifies the mapping of object names onto host addresses. An example of such a naming mechanism can be found in the distributed file system of the RSEXEC,[75] which was developed more than a decade ago for the TENEX hosts on the ARPAnet.

2. *A system name is a globally unique name of a process.* This naming scheme is already much better. User processes may augment system names with port names, which can be interpreted by server processes as object names. The problem with this method is that a service or an object usually lives much longer than a process. On top of process naming, a level of naming is needed where service names are mapped onto process names. This solution usually leads to a (centralised) name server, which needs a well-known, never changing process name, since changing the name server's name means that its clients are no longer able to address it. If the name server crashes, a new one must be created with the same process name; the operating system, which gives out process names, must see to this; it has to deal with the name server as a special case.

   An example of a system that uses process naming is the National Software Works.[44] Another example is the DCS,[23] which is of special interest to us, because it uses a mechanism for locating 'ports' that is rather similar to the one described in § 2.5.1.

3. *A name is a port that can be allocated to processes by the Operating System.* There are several examples of distributed operating systems based on this principle. In this model, ports can live much longer than processes. Ports can be associated with services, and the server processes form the receiving end of such ports. When a process crashes, another process can take its place, using the same set of ports. Processes wishing to connect to a service, connect to the associated port, and transmit requests to whichever service process listens at the other end of the port. The name of the service is the name of the port.

   There are several examples of distributed operating systems using this model for interprocess communication, as for instance, *Trix*,[78] which uses named *streams* as interprocess communication primitive. A stream has an owner (the service process) and one or more holders (the clients). Holders and owners may pass their ends of a stream to other processes. There can be only one owner of a stream. A similar mechanism is present in the *links* of the Roscoe distributed operating system,[67] the *streams* of Accent[53] and the *ports*, described by Akkoyunlu et al.[2]

4. *A name is a port, created and managed by user processes.* In this model, the processes of the system can generate their own ports, with whatever name they care to

choose. When a process decides to offer a service to the other network users, it generates a port for that service, publishes the port's name for potential clients of the service, and starts listening to that port. Clients then send messages to the service port, which will be received by the server process. When a server process crashes, or the work becomes too much for one server process, a new server is started, which listens to the same service port. The operating system (if there is one) need not do a thing. The network sees to it that messages sent to a port are delivered to a process listening on that port.

Long-lived services are easily implemented using this message passing scheme. Even if the service outlives all its servers, the name of the service can remain unchanged. Processes, wishing to communicate once for a short period of time use the same mechanism: a unique port is generated and used for the duration of the conversation. When the conversation is finished, the port can be forgotten.

So far, so good, but this communication mechanism must be secure if it is to be used in a distributed operating system. What, for instance, is there to prevent a client process, which can transmit requests to the server's port, to start receiving messages on that port, thus intercepting requests from other clients? And how can we guarantee that no two processes accidently choose the same port name for their conversations? In the following sections we shall discuss a mechanism that achieves these ends and describe how it can be implemented efficiently.

2.2. Protection and Reliability

A good interprocess communication mechanism must provide both reliable and secure message transport. Reliability and security are, however, achieved in two different layers of the interprocess communication protocols. In most distributed systems, reliable communication is implemented at the lowest layers of the interprocess communication mechanisms. Protection is usually found at a higher level in the hierarchy.

We have turned the usual layering upside down. Protection is implemented in the bottom layers of the network protocols, and reliability (through acknowledgements, timeouts and retransmissions) comes in a higher layer of the communication hierarchy. We have done this for serious reasons, into which we shall now go.

It is far easier to build a secure network than a secure operating system. Experience teaches that only very few operating systems are free of bugs which allow an intruder to steal secret information from it. The communication subnet can be made secure much easier, because it can be better isolated from the rest of the system: only a narrow interface exists between processes and the network. Operating systems have many explicit interfaces (such as system calls) and implicit interfaces (program faults, memory management, i/o buffers) with user processes, giving many points where the security of the operating system could be penetrated.[34] The protection mechanism is far safer where the user cannot get at it.

Another argument for removing the protection mechanism from the operating system is that some (micro)computers do not have the hardware necessary to build a secure operating system. When no memory management is present allowing certain areas of memory to be protected, it becomes very difficult, if not impossible, to build a secure operating system.

A last point for moving protection out of the operating system into the network, is that it is far easier to tamper with the hosts of a network, than with the network itself. Imagine an office building, with a microcomputer in each office, and possibly some centralised facilities, which can be used by the micros. An intruder—late at night, for instance—can stop one of the micros, insert a specially prepared operating system disk and steal all kinds of secret information. The point here is that it is almost impossible for a network authority to ensure that the correct operating system is run all the time when the hosts are not all under his direct supervision. A network can be much better physically protected, for instance, by keeping the network nodes in secure places, or—as an extreme resort—by making the network interfaces give off an alarm when they are tampered with.

It is for these reasons that the Port Layer, the layer that provides protection mechanisms is under, or—if necessary—in the operating system, and the Transaction Layer, the layer responsible for reliable transactions rests on top of the Port Layer, in the operating system, or even in user space.

2.2.1. Protection Policies and Mechanisms

The protection mechanisms must be chosen to make a wide choice of protection policies possible. Separation of mechanism and policy has been advocated by Brinch-Hansen,[6] and we believe it is good idea. It allows policies to be changed, without changing the mechanisms embedded in lower layers of the system. We shall look at some of the policies that we think should be made possible by the protection mechanisms.

Most distributed systems use the notion of **services**, an abstract notion of an entity that executes requests for the users of the system. Every service has a different set of commands that it is willing to accept and execute. A service can give access to **objects** that it manages for the owners. This makes the principle of services much like an implementation of abstract data types.[30] But in most distributed systems the principle of addressing an abstract service is not reflected in the design of the interprocess communication mechanisms. Our interprocess communication mechanism should support the policy of addressing services and objects managed by these services, but also allow a process addressing policy, or even a machine addressing policy.

Processes must not be able to send messages to every service. Some services are private (*e.g.*, the execution of private program can also be considered a service), some services are public (*i.e.*, every process may use it), and some are semi-public; that is, they may only be used by certain processes or devices (*e.g.*, it could be a policy to allow only certain file servers and data base servers to have access to the disk server). We suggest capabilities can be used to prove a process' permission to send to a service, and likewise capabilities can be used to prove the right to receive messages addressed to a service.

The right to send messages to a service does not imply the right to receive messages sent by other processes to that service. It would enable a process with a capability to send to a service to impersonate that service by receiving its messages.

Since the receiver of a message must be able to tell who sent it, the option to authenticate the source of a message should be present. This authentication must be unforgeable, and a process that receives an authenticated message must be able to verify the message's source, but may not be able to use the authentication for its own messages. It is of course a matter of policy whether and how the authentication mechanism will be used.

2.3. The Protection Mechanism

In this section we introduce an interprocess communication mechanism that can be made secure with the use of a secure network interface, the **F-box**, but does not require a secure operating system on the hosts. The interprocess communication mechanism rests on a capability-based naming mechanism—capability based, because capabilities are needed to send and receive messages. In our system, capabilities consist of knowledge, knowledge of a **port**, a large random number. There are two kinds of ports, **get-ports** and **put-ports**. *Get-ports* are capabilities for the reception of messages. *Put-ports* are capabilities for the transmission of messages. Every *get-port* has a *put-port* associated with it. Messages sent to a *put-port* are received by processes listening to the associated *get-port*.

In our communication model, capabilities consist of knowledge. A capability to send to a port, or a capability to receive on a port is just a large number, and knowledge of this number is taken by the system as *prima facie* evidence of the right to use that port. A capability can thus be passed to other processes, by just wrapping it up in a message and sending it. There need not be any 'pass-capability' operation in the operating system. If knowledge of ports is enough to use them, some care must be taken to prevent capabilities from leaking out accidently. In particular the network must be secure against eavesdroppers, and multiprocessing systems in ***Amoeba*** must be secure to prevent processes from stealing capabilities from other processes on the same host. Monoprocessing systems however, need not have an operating system at all, since the network will not deliver messages if the receiving host can not show a capability for them.

2.3.1. Ports and One-Way Ciphers

Ports are random numbers—large random numbers—which ensure that the space of all possible port names is very large compared to the space of the actual port names used; the port name space is thus sparse. Through sparseness it is possible to use knowledge of a port name as a capability for that port. Otherwise it would be possible to generate random port names with a reasonable chance of generating an existing port name. In a large network with 2000 processes and an average of 5 ports per process, a fully random 48-bit port could be broken by brute force in 2.8×10^{10} tries, on the average. At a rate of 50 tries per second, it would take almost eighteen years of continuous trying to find just one port.

Different capabilities are needed to send to a port and to receive on a port. The send capability is the **put-port**, and the receive capability is the **get-port**. Every get-port is related to a put-port by the following relation:

$$\textit{put-port} = F(\textit{get-port}),$$

where F is a **one-way** function.

A one-way function maps a number, the **plaintext**, onto another, the **ciphertext**,* with the following properties:

- Given x, it is easy to compute $y = F(x)$.
- Given $y = F(x)$, it is infeasible to find x.

Note, by the way, that is *not* required that an inverse function, F^{-1}, exists such that $\textit{get-port} = F^{-1}(\textit{put-port})$. In fact, it is required that F^{-1} either not exist at all, or at least be too cumbersome to use. In this respect one-way functions differ from cryptographic functions.

We shall not attempt to quantify *easy* and *infeasible* exactly, because these concepts vary, depending on the application, the amount of security required, and what is technically possible. To give a rough idea, for ***Amoeba*** we are thinking in terms of 'encryption' times of in the order of 10 to 20 msec., allowing encryption at approximately message transmission times, and cracking times on the order of at least ten years. The principle of one-way functions is due to a password encryption system by R. M. Needham, and was first mentioned by M. V. Wilkes.[80] In a later section we shall discuss possible functions suitable for our purpose.

Let us see how ports and one-way functions are used in interprocess communication. First a service S chooses a port name, P_g (using a good random number generator). P_p is then computed, by applying F to P_g. The put-port, P_p, is given to 'the public,' the potential clients of the service, and P_g, the get-port is

* The terms *'plaintext'* and *'ciphertext'* are not used here as in cryptography: ports are converted from plaintext into ciphertext, but it is intended to be impossible to convert a ciphertext port back into a plaintext port.

kept carefully secret and given only to the processes constituting the service. Suppose that S' is a server process for service S. S' announces to the network its intention to receive messages on get-port P_g (in ***Amoeba*** terminology it does a *get* on port P_g). The F-box converts P_g to P_p, by applying F. Let us assume process A wants to make use of the services of S. It could obtain the capability for S from a 'yellow pages' file, a file containing a list of services and service ports. A then sends its request to S, using P_p as the address (in ***Amoeba*** terminology, A does a *put* on port P_p). The network delivers the request to one of the (possibly many) processes with a *get* on port P_g, by comparing the destination port in the messages to the put-port derived from P_g, P_p. Requests from A can thus be delivered to S'.

The application of one-way functions to ports makes it possible to give a capability for communicating with a service to a process without giving the process the chance of impersonating the service. We must assume that the network is secure; that is, an intruder cannot listen to the traffic passing through the network, because that would give the intruder the chance of obtaining put-ports. Many put-ports will be public (*e.g.*, the put-port for file-service, or time-of-day service), but many more put-ports are secret, namely all put-ports used in communication between private processes, or possibly in the communication between a distributed file server and its disk drivers.

2.3.2. Reply Ports and Signatures

The mechanism described fulfills only partly the requirements for authenticated communication: it was shown how the receiver of a message must provide an authenticating capability for receiving messages, but the mechanism that prevents a client or service from sending messages that purport to originate from another client or service has not been shown yet. It is highly unsatisfactory if the communication between a client and a server can be disrupted by a process sending phony acknowledgements for that client to the server. To prevent this, we use F, our one-way function, to transform the reply-port in every message before it is delivered to its destination. This means that requests can only be sent by processes with a capability for receiving the reply. An intruder who happens to know a process' put-port can otherwise cause a totally innocent server to send messages to that process, by sending messages to the server with that put-port as the reply port. If the process were already in communication with the server, it may thus become confused.

The same mechanism can also be used to *sign* a message. This is especially useful for services that have to rely heavily on the identity of their clients. Bank Service (presented in Chapter 7), is a good example of such a service. Every user of the system can choose one or more signatures, register them with the proper authorities (depending on system policy; registration could be with an authentication service in one system, and with every separate service in another), and use the signature to send signed messages to servers. The signature cannot be misused by the processes that receive them, since only the private plaintext-signature can be used to sign a message. The ciphertext-signature can be public and be used to check the authenticity of the sender. In Chapter 7 more will be said on the subject of signatures.

2.3.3. One-Way Functions

One-way functions have been studied by a number of people who have used them for password protection in operating systems. Basically there are two different approaches to designing a secure one-way function; the first is finding a function that is mathematically hard to invert, the second is to find a function that is computationally hard to invert. The two methods are different. In the first method we try to find a function about which we can prove certain properties, one of the properties being the amount of work involved in computing the inverse function. In the second approach, we try to find functions of which we do not know the mathematical properties, but can be confident that the computational difficulties involved in finding the inverse are immense.

The **degeneracy** of a one-way function is the maximum number of x's mapped onto any one $F(x)$. The degeneracy of one-way functions is important, because the greater the degeneracy, the greater the chance of breaking a one-way function. Suppose a one-way function maps the numbers $\{0, \cdots, N-1\}$ onto $\{0, \cdots, N-1\}$, with degeneracy d. For any one $y \in \{0, \cdots, N-1\}$ there can be as many as d numbers x, such that $F(x) = y$. The probability of breaking a one-way function by searching for a plaintext that produces the desired ciphertext is d/N per probe, that is, the probability of breaking a one-way function by brute force is directly proportional to the degeneracy.

Most one-way functions of mathematical origin are based on properties of natural numbers, derived from number theory. One possibility is to use a polynomial to a prime modulus, as described by Purdy:[52]

Choose a one-way function $F(x)$ by choosing a polynomial

$$p(x) = x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_{n,}$$

where n, $a_1, \ldots, a_n$ are integers. Let P be a large prime number, and define $F(x)$ by

$$F(x) \equiv p(x) \pmod{P}$$

Then $F(x)$ can be used as a one-way function that maps the numbers $0, \cdots, P-1$ onto $0, \cdots, P-1$. Polynomials of degree n to a prime modulus have n roots or less [*cf.* 39, theorem 3.15, p.42]. Consequently, the degeneracy of F, which is the maximum number of x's mapped by F onto any one y, or the number of solutions to the congruence

$$F(x) - y \equiv 0 \pmod{P}$$

does not exceed n.

Given that $y = F(x)$, breaking the function amounts to finding roots of the congruence $F(x) - y \equiv 0 \pmod{P}$. This can be done by brute force, that is, by trying random ports, applying F to them and checking if they produce one of the desired cipher-ports, or it can be done by finding the inverse function of F using a *polynomial root finding algorithm*. The best root finding algorithms to date require $cn^2(\log P)^2$ operations to find the roots of a polynomial of degree n, modulus a prime P; c is a constant.[32,4] The brute force approach (trial and error method) requires an expected number of P / nm tries to find an inverse of one of a set of m ports. Suppose, for example that $P = 2^{64} - 179$ (which is prime), $n = 2^{19} - 1$, and the number of ports in the network $m = 10{,}000$. Then the number of operations required to break the function by the root finding method is on the order of $n^2(\log N)^2$, or 5×10^9. At a rate of a million operations a second, it would take about 17 years to break the function using this approach.

The expected number of tries needed to break one port of a set of m cipher-ports is P / nm, or 3×10^9 when $m = 10{,}000$. Assuming it takes 10 msec to compute $F(x)$, it would take 400 days of CPU time to find just one get-port out of a set of 10,000, and even then, chances are that the port thus obtained would be no use whatsoever to the cracker.

As an example of a computationally hard to invert one-way function consider the work of Evans et al.,[22] who, in their proposed one-way function, use a small set of one-to-one mappings, which eventually produce a ciphertext. In a nutshell, their scheme works as follows: Assume the one-way function must map n-bit *get-ports* onto n-bit *put-ports*. Let $f_1, f_2, \cdots, f_k$ be k different scrambling functions. Each of the f_i must be one-to-one, to keep the degeneracy within bounds. The scrambling functions map an intermediate result X of the calculation into a new value of X'. Some of the functions may be parameterised by the original plaintext-port P, or a small integer, m, derived from P or X. Possible scrambling functions can be:

- rotate m bits
- permute bits, fixed or calculated (from P or m)
- compute the binary sum (modulo the size of P and X) or the XOR of P and X.
- add a constant, or a quantity calculated from P to each byte of X.
- choose some of the machine instructions that can be used to perform some one-to-one transformation of X, parameterised by P or m. Use a subset of these, calculated from P.

FIGURE 2.1 shows how these functions are then used to transform P into a cipher-port V, via a succession of Xs. K is the number of different scrambling functions, J is the number of cycles, and the function *next* computes new values of X from old. The function q_k derives small integers from P and X.

The result of the one-way function thus obtained, is very hard to analyse, and the degeneracy is probably very small.[22] Each of the f_k is one-to-one. The composition of one-to-one functions is also one-to-one, but Evans' algorithm is not an ordinary composition of the f_k, but one depending on the value of P, which makes it possible that it is no longer one-to-one. This type of one-way functions

```
#define J NumberOfCycles /* e.g., 5 */
#define K NumberOfDifferentScramblingFunctions

port F(P) port P; {
      port X, V;
      int j, k, m, M;

      X = P; V = P;
      for(j = 0; j < J; j++) {
            for(k = 0; k < K; k++) {
                  M = q_k(P,X);
                  for(m = 0; m < M; m++)
                        V = f_k(V,P,X);
                  X = next(X);
            }
      }
      return V;
}
```

FIGURE 2.1. Calculation of Evans et al.'s one-way function

is most suitable for our needs: the degeneracy is small, so a trial and error approach will take a very long time to yield results, and other approaches to cracking the function are not known. The function can be chosen such that it takes, for instance, 20 msec to compute, limiting the number of tries in an attempt to break it to 50 per second. At this rate, with a degeneracy of say three, it will take 6 years on average to get just one plaintext-port of a set of 10,000 48-bit ciphertext-ports, not counting the time needed to look up the result of the encryption to see if a port has been cracked. With a 64-bit port, the time is at least 256 times longer. It is, as with any cryptographic system, possible that structural weaknesses may make it easier to break the one-way function.

Another well known approach to finding computationally hard to invert one-way functions, is to use a cryptosystem as a basis. Good results can be obtained by using the number to be enciphered as the key to encrypt a known constant. Consider, as an example, the *Data Encryption Standard*, DES.[69] The key for this cipher is 56 bits, and it is used to encrypt or decrypt 64 bit quantities at a time. DES can be turned into a one-way function by encrypting a known 64-bit constant, using the plaintext as the key.

2.3.4. Secure Communication in an Insecure Network

In the one-way function scheme we have assumed that the operating systems in the hosts are insecure, but that the network is secure. In many local networks this demand can be met, but it is conceivable that networks exist that cannot be made secure. One solution to achieving more security is to use **link encryption** between the network nodes. This prevents information from being stolen by wire tapping, but the network must still guarantee application of F to get-ports, reply-ports and signature-ports. If the network cannot even guarantee that, another solution to achieving secure communication and protection is needed. Fortunately, a solution can be found in a public-key encryption scheme[16] which can be used to build a secure network layer, with an identical interface to higher layers.

The public-key encryption scheme amounts to this: the public key takes the place of the put-port, and the private key that of the get-port. When a service is created, a public-key pair is generated, plus an identifying port number. The port number is used to recognise which messages should be associated with which service, and need only be sparse enough to prevent processes from accidently choosing the same service number too often. When a message is sent from a client to a server, it is encrypted with the public key, which serves as a capability for addressing the service intelligibly. In principle the message can be intercepted by all the processes in the system, but only a process with a capability for receiving the message—the private key—can understand the message. In order to avoid tampering with messages, each one must contain an encrypted checksum which also prevents interpretation of nonsense messages. A timestamp can be included to prevent playback of previously intercepted valid messages. A port in this public-key scheme consists of an identifying number, plus a key, a public key for put-ports, and a private key for get-ports.

In wide-area networks, public-key cryptography is already being used. The encryption speeds are in the same order of magnitude as transmission rates in wide-area networks. Public key ciphers are still too slow to be generally used in local-area networks. However, we expect that as soon as public-key hardware comes on the market, it will be possible to encrypt messages as fast as they are sent. Another problem of public-key ciphers is that public-key ciphers with reasonably short keys do not yet exist. Current cryptographic algorithms use keys of hundreds of bits. Perhaps, cryptographic research will yield algorithms with much shorter keys some day; not all public-key schemes need use factoring.

2.4. The Port Layer

The conglomerate of activities related to ports is referred to as the **Port Layer.** In functionality it comes closest to the *Network Layer* of the ISO OSI model. The Port Layer itself is not normally visible to user programs, it must viewed as a layer in the operating system, or as a hardware device, that supports the Transaction Layer (the implementation of which is also inside the operating system). User programs (and most of the operating system itself) see only the Transaction Layer interface.* The Port Layer provides a secure datagram service to the

* See also Chapter 4, where the operating system interface is described.

layers above; that is, the Port Layer provides protection through ports, it locates ports and it delivers messages, addressed to ports. The Port Layer only provides a datagram service; it does not guarantee the delivery of every message, although it makes an effort to do so; nor does it provide sequencing, *i.e.*, messages need not arrive in the order they are sent; in short, it does not provide a byte stream protocol.

There is one vital difference, however, between the datagram service provided by the Amoeba Port Layer, and most other datagram services: the Port Layer does guarantee that messages never arrive more than once: *messages arrive once, or not at all*. This is realised because the Port Layer never retransmits messages, unless it 'knows' that the message has failed to arrive. Furthermore, messages are never broadcast, as this may result in two or more hosts, listening on the same port, receiving the message.† In § 3.2, which describes the Transaction Layer, we shall show that this property simplifies the Transaction Protocol considerably.

The next four sections describe the interface to the Port layer and its three functions: protection, locating ports, and delivering messages.

2.4.1. The Port Layer Interface

The Port Layer interface provides calls to send and receive *messages*, which have (almost) unbounded length (currently 32 Kbytes). If necessary, these messages are split up into small units and sent piecemeal. A message consists of two parts, a 40-byte *header part*, and a *data part*. The structure of the header is shown in FIGURE 2.2; the header consists of a *length* field, giving the size the data. A header-only message is 40 bytes long, and has a *length* field of zero. Then comes the *destination port*, the *put-port* for the port where the message has to go. The third field is the *reply port*, the *put-port* where reply messages are to be sent. The next field is the *signature port*, also transmitted as a *put-port*. 20 bytes are left over for use by higher layer protocols.

The most important calls are the calls for sending and receiving messages, *get* (or *getany*) and *put* (or *putany*). These calls may or may not be blocking, depending on the structure of the kernel. In the current implementation of the Amoeba Kernel,* *get* and *put* are non-blocking and implemented in the lowest layer of the kernel for best performance. Each call has two parameters, pointers to two buffers, one for the message header, and one for the message body. For transmission and reception of header-only messages the buffer pointer can be a dummy parameter. In a *get* call, the pointers point to two buffers for the message to be received. Only three header fields must be filled in by the receiver. The length field must be set to the length of the buffer available for the data portion of the received message, the destination port field must be initialised to

† With the exception of special ring networks as described in § 2.5.2.

* The Amoeba Kernel is described in Chapter 4.

Message length
Destination port (6 bytes)
Reply port (6 bytes)
Signature port (6 bytes)
Out-of-band data (20 bytes)

FIGURE 2.2. The Port Layer header layout

the *get-port* on which the message is to be received, the reply port field is set to the sender's *put-port*. A *get* call is thus a call to receive messages from a specific source. It is also possible to receive messages from any source using *getany*. The only difference between *get* and *getany* is that the reply port need not be specified in a call to *getany*.

On a *put* call, the two pointers point to the header and buffer to be transmitted. The header fields are filled in as follows: The length field is set to the length of the data buffer to be sent. The destination port field is set to the *put-port* for the receiving port. The reply-port field is set to the *get-port* for return messages. This port is converted by the F-box to a *put-port* before the message is copied to the net. The signature port field is set to the plaintext version of the optional signature to be sent. The F-box transforms the signature to a ciphertext version, using the same conversion as for ports. The remaining 20 bytes of the header and the data buffer can be filled as the sender sees fit.

A message, transmitted via *put* can only be received by a process with an outstanding and matching *get*. Messages sent using *putany*, can only be received by processes with an outstanding and matching *getany*.

A message, sent with *put* matches an outstanding *get* when both the destination ports and reply ports match. A message, sent with *putany* matches an outstanding *getany* when the destination ports match.

A few other calls are available to users of the Port Layer, *unget* and *local*. *Unget* is used to cancel outstanding *get* or *getany* operations, *e.g.*, when a message is no longer expected. Its single parameter is the address of the header of the outstanding *get* to be cancelled. *Local* has two parameters, a *get-port* and *flag*. Depending on the value of the flag, it tells the system that a port is local to the calling process, that the port is local to the caller's host, that the port is local to the caller's local-area network, or that the port is global. A new call to *local* cancels the effect of a previous call with the same port.

Summarising, the calls are:

```
get(&header, &buffer);
getany(&header, &buffer);
put(&header, &buffer);
putany(&header, &buffer);
unget(&header);
local(&getport, howlocal);
```

A *get* call terminates on reception of a message, or is terminated by a call to *unget*. When a message is received, the *length* field is adjusted to the message's length. Messages that do not fit into the buffer are truncated, and the recipient is notified.

As far as the Port Layer is concerned, a *put* call terminates successfully when the message is sent. Note that this is no guarantee of the message arriving. *Put* terminates unsuccessfully if one of its parameters is in error, or when the destination cannot be found; *i.e.*, the *locate* fails.

Port Layer messages can be very long. Most local-area networks are not capable of sending such messages as one network message. However, by optimising the Port Layer for optimum performance on the local-area network it runs on, transmission speeds can be obtained that could not possibly be obtained using other techniques (such as sliding window protocols). Memory is getting cheaper every day, and modern microcomputers often have very large address spaces, so room to store these large messages can usually be found. Note, by the way, that most messages are short,* and that communicating parties have ways of bounding the lengths of received messages (*e.g.*, by making requests for small portions of data).

2.4.2. Protection via One-Way Ciphers

The Port Layer provides protection, using one of the protection mechanisms described earlier: one-way functions, or public-key-based protection. To make this work, it must not be possible to circumvent the Port Layer protocols, nor must it be possible to tamper with them. In traditional operating systems, the Port Layer would have been built into the operating system kernel, safe from attack by ordinary user programs. Operating system kernels, however, are seldom without security loopholes, due to their enormous complexity and their elaborate interfaces.

Distributed operating system kernels are even more vulnerable to attack. Often, a distributed system consists of a cable running through the building, with personal computers, file servers, etc. plugged in all over. It is impossible for a central authority to keep an eye on every host of the system, so a malicious user can easily replace the 'official,' secure operating system kernel by one without

* See § 3.2.1.

any protection mechanisms.

As stated before, this has been the prime motivation for turning the usual layering of interprocess communication services upside down, and provide secure communication in the bottom layers. This makes it possible to implement the protection mechanisms in hardware, or to put them in the network nodes, or in the network interfaces.

This is not the only reason for putting protection in the bottom layers of the communication hierarchy, however. When protection is in a lower layer, higher layers can be designed on the assumption that the communication between two parties is completely private: there can be no intruders to disrupt communication by inserting phony messages, replaying messages, etc. The protocols needed for reliable, sequenced communication become much simpler this way.

Conceptually, the Port Layer is implemented in a 'device,' inserted between a process and the network. We refer to this device as an **F-box**, for *function-box*, because its most important task is to carry out the protection mechanism by application of the one-way function.

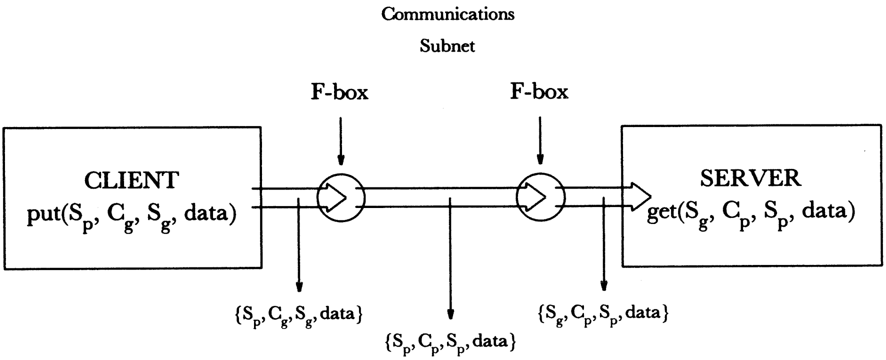


FIGURE 2.3. A local network with hosts and F-boxes.

The F-box applies the one-way function to *get-ports, reply-ports* and *signatures*, as illustrated in FIGURE 2.3. It can be a microcomputer, inserted between a host and the network, it can be built into a network interface board, it can be part of the host's operating system, or it can be part of the software of the network nodes in a mesh network. Whatever solution is chosen, it is secure only if every *put* and every *get* pass through at least one F-box before a *rendezvous* is made, and furthermore, if the F-boxes cannot be tampered with.

Many local networks use a coaxial cable that runs through the building with an outlet in each room. People who want to use the network can plug their intelligent terminals, word processors, minicomputers or microcomputers into the wall and use the services provided via the network. The F-box can be built into the outlet in the wall, where it can be made safe from tampering. This system allows connection of any hardware to the network, as long as it is capable of communication with the network interface. F-boxes can also be built into the network interface board. This is not as safe as putting the F-box in a tamper-proof outlet in the wall, but it prevents circumventing the F-box, unless the hardware is tampered with. More secure than putting the F-box on the network interface board, is to build an F-box into a network interface chip. To circumvent such an interface requires building one's own network interface. From the point of view of physical protection, protection by means of hardware F-boxes is preferable to a protection mechanism in the operating system of the hosts. Tampering with an operating system is orders of magnitudes easier than tampering with electronic hardware. Electronic tampering is also easier to detect.

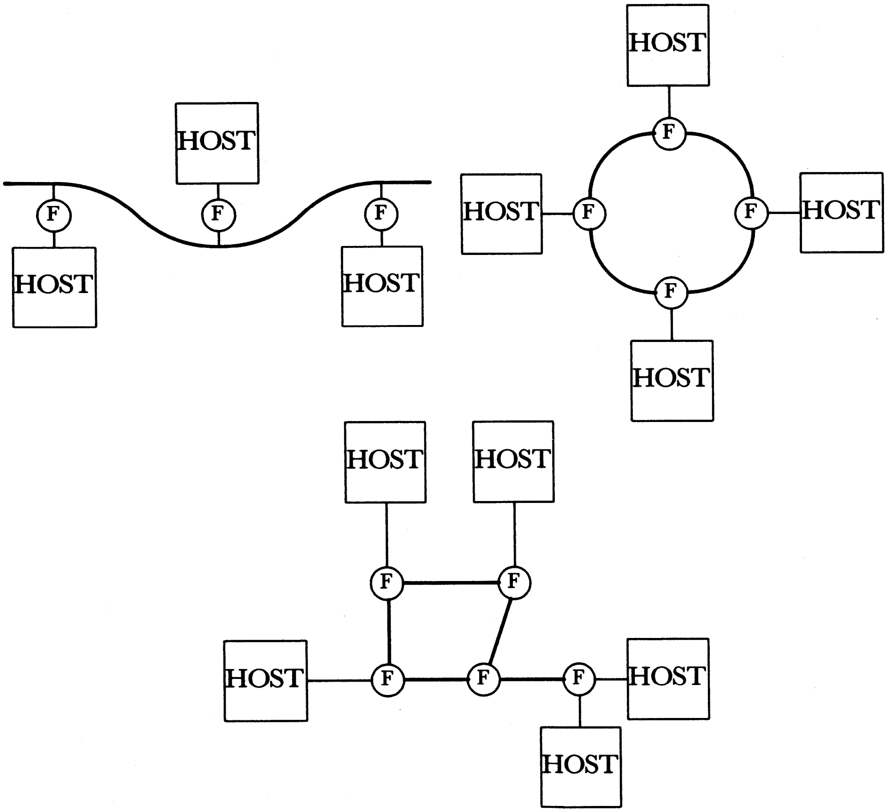


FIGURE 2.4. The structure of several kinds of secure networks, with F-boxes inserted in various places.

In a mesh network, every host is connected to a network node. A message travels from one host to another via one or more of the network nodes. These network nodes are usually small computers, under control of the network management, executing secure network code, which makes them eminently suitable to carry out the one-way functions. FIGURE 2.4 shows some examples of network structures with F-boxes.

There is thus a close relation between the Port Layer and the F-box: the Port Layer is the collection of protocols that provides a secure (but not necessarily reliable) datagram service to higher layers; the F-box is the device that carries out those protocols. The terms *Port Layer* and *F-box* will often be used interchangeably.

2.5. Locating Ports

The ***Amoeba*** distributed operating system uses *ports* to address messages. When a process sends a message to a port, the system must find the location of another process receiving on that port in order to deliver the message. In this section, some methods are discussed to locate ports so that messages may be delivered. These methods need not apply solely to locating ports, but they are probably equally applicable in other systems where, given a location-independent name for an object, the place where it resides must be found.

The Port Layer provides two sets of primitives for sending and receiving messages. One set provides primitives where a source process does not care which of the destination processes listening on the destination port receives the message, and the destination processes do not care which source process sent the message. The second set provides primitives where a source process sends messages to a destination process listening specifically to messages from that source. In the first set of primitives only the destination port must match between sender and receiver. In the second set of primitives both destination port and reply port must match.

This has consequences for locate operations. There must be two kinds of locate, one to find a recipient for messages sent using *putany*, and one to find a recipient for messages sent using *put*. In the first case, locate answers the question 'who's listening for messages for destination port x?' In the second case, it answers 'who's listening on port x for messages from port y?' The mechanisms for answering these questions are the same, so the difference between the two kinds of locate is ignored in the remainder of this section.

Different types of network require different techniques for locating ports. In some networks, such as a star network, locating a port is not difficult: the central network node must be provided with a table of ports and their locations, and it can do the trivial routing of messages. In other networks, locating may be quite complicated: a hierarchy, for instance, of several network types, connected together by an internet. In this section, two types of network will be discussed, broadcast networks, and ring networks equipped with special hardware. In § 2.6, methods for locating ports in different types of mesh networks and network hierarchies will be discussed.

2.5.1. Broadcast Networks

Two network types have become popular in local-area networks, carrier-sense networks and various kinds of ring networks. Both types are broadcast networks; messages are broadcast on the network medium (a co-axial cable, twisted pair, optical fibre), each of the hosts' network interfaces examines passing messages and reads those messages intended for it off the cable. Usually there is a *broadcast address*, an address that all interfaces react to, so messages can be broadcast using this address.

Naturally, the ability to broadcast messages on the network helps in locating ports. When a node wishes to send a message to a port whose location is unknown, it just sends a *'where is port such-and-such'* message to all other nodes. The node where the port resides can reply with a *'port such-and-such is at node so-and-so'* message.

Note that when the message itself is broadcast, it may be received more than once if two or more network nodes receive on the same port, which is not uncommon when a service is offered by more than one server process. Another problem with broadcasting (large) messages rather than tiny *where-are-you* messages is that each node must reserve enough buffer space to receive those messages only to discover they have to be discarded again, because they are for another node.

To make port locating efficient, each node maintains a cache of known port/host-address pairs. (Since there are two kinds message passing—from a specific source or from any source, there are two kinds of port entries in the cache: single ports and double ports.) The rules for updating and using the cache have to be set carefully: on one hand, messages should nearly always be delivered correctly, even when ports move from one host to another, or when a port is no longer active on one host, but still being listened to on another; on the other hand, under no circumstances may messages be delivered more than once, since higher level protocols rely on the property of the Port Layer that messages are delivered exactly once or not at all.

A set of rules that does the trick uses four types of messages:

1. broadcast message *'where is port* x*?'*
2. point-to-point message *'port* x *is at node* y.*'*
3. message for port x at node y.
4. rejected message for port x by node y (port x not at node y).

These messages are used as follows:

When a message is given to the Port Layer for transmission, the cache is examined for presence of the destination port/host pair. If it is there, the message is sent to the node, indicated by the contents of the cache, so suppose it is not there. Then a *'where is port* x*?'* message is broadcast and a timer started. When a *'port* x *is at node* y.*'* message arrives, x and y are stored in the cache, the message is transmitted, and the timer is stopped. If the timer expires before any replies have come in, the message is discarded, and failure is reported to the sender.

When a *'where is port* x*?'* message is received, the table of listened-to ports is examined. If an entry is found that matches the request, a *'port* x *is at node* y.*'* is returned. When a message for port x is received, and an entry for x is found in the listened-to-port table is found, the message is delivered to its destination. Otherwise, the message is returned to the sender as a *rejected message for port* x *from node* y.*'* When a node receives this last type of message, it must erase the corresponding entry from its cache, and treat the message as if it were a message of local origin to be delivered to port x.

Note, that a node, sending a message, may discard the message immediately after transmission; if the message fails to arrive because the information from the port cache is no longer up to date, the message will be returned, so it can be retransmitted. In most cases, the overhead of returning the message as a whole, instead of a small control message, is more than compensated by not having to remember every message just in case it is rejected at its destination. Network bandwidth is cheap in modern local-area networks.

It is not difficult to verify that messages can never be accepted more than once, when this set of rules is used. When no messages are lost in the network, and the destination host is up, the contents of the cache never cause messages to be lost. When the F-boxes are constructed in such a way that the F-box continues to work if the host crashes, no messages are lost, even when the destination host is down. However, messages may not be delivered then, because no alternative port is present to send the messages to.

2.5.2. Special-Hardware Ring Networks

With some small modifications, token rings, contention rings and slotted rings[72] can be used for an especially efficient implementation of the Port Layer functions. No caches are needed; no host need know where ports are, except, of course, the ports listened to by processes in that host itself. Before we describe the modifications and the protocol, let us describe the properties of ring networks that make the protocol work.

In a ring network, each message is put on the ring bit-for-bit by the sending node. Each node on the ring, including the destination node, copies the bits on the incoming line from the previous node to the outgoing line to the next node. A message thus travels all the way round the ring back to the originating node. The destination node alone not only copies the bits to the outgoing line, but also copies the bits into its own memory, so it can deliver the message. Often, one bit in the header or trailer of each message is not copied by the hardware in the receiving node, but set. This is the *accepted* bit, and it tells the sender that the message has been received. This bit is going to play an important role in our protocol. We shall now describe the proposed modifications to the hardware.

The first modification concerns the address recognition mechanism in each node. Normally, an address consists of, say, eight bits and is wired into the network interface. The first bits of a message form the address, and while the bits are copied through, the network interface compares them to the node address. If they match, the rest of the message is also copied into memory. In the modified approach, addresses are no longer wired into the interface, but a table of addresses is put on the interface. The entries of the table are ports and the contents of the table can be changed by the operating system.* When a message passes by, the address at the head of the message (a *put-port*) is collected, and compared to the contents of the table. This comparison takes time, so the contents of the message are copied into a buffer on the interface board while the comparisons are being made. If the port is not found, the contents of the buffer can be discarded, and the rest of the message is copied through unmodified. If the port is found in the table, the rest of the message is copied into the buffer, and the *accepted* bit is set when it passes by, noting the old contents of the bit, however. If the *accepted* bit was already on, a previous node already had accepted the message, and, since messages may not be received twice, the contents of the buffer are discarded. If it was off the message had not yet been received, so the message is accepted and delivered to the host. (The distance in a message between a port and the *accepted* bit must be large enough to make this possible. It is a fortunate co-incidence that the *accepted* bit is usually in the message trailer.)

2.6. The Shotgun Method for Locating Ports in Mesh Networks

Before a client can send a request to a server which provides the desired service, the client has to locate that server. The problem of efficient *routing* arises at a later stage; first the address of the destination has to be found in a **match-making** phase. Match-making is finding the address of a server (or any object, for that matter), given its name. A match-making service is often referred to as a **name server**.

A *centralised* name server must reside at a so-called **well-known address** which does not change and is known to all processes. (Clearly, the name server cannot be used to locate itself.) When the name server crashes, or its host, the entire system crashes. This solution also causes an overload of messages in the neighbourhood of the name server.

When clients **broadcast** for services with "where are you" messages, we have an example of a **distributed name server**. This solution is more robust than the centralised one, and practical in most local-are networks, which are usually broadcast networks anyway. But in large store-and-forward networks, where messages are forwarded from node to node to their destination, broadcasting is considerably more costly than sending a message directly to its destination.

* When the operating system writes a *get-port* into the table, the F-box hardware on the network interface applies F to the port before storing the resulting *put-port* in the table.

Broadcast messages are sent to every host, while point-to-point messages need only pass through the hosts on the path between client and server. Conventional broadcast methods for locating services need a minimum of $\Omega(n)$ message passes to do the broadcast (*e.g.*, via a spanning tree[13]).

In this section, realisations of name servers are investigated in the entire range between centralised and distributed forms. The efficiency of solutions is measured in terms of message passes and local storage. It appears that, in many n-node networks, very efficient distributed match-making between processes can be done in $\Theta(\sqrt{n})$ message passes.

2.6.1. Locate Algorithms

In all cases, the method used to locate a port is the following: A server process s located at address A_s and offering a service identified by a port π, selects a collection P_s of network nodes and **posts** at these nodes that server s receives requests on port π at the address A_s. Each of the nodes in P_s stores this information in a cache for future reference. When a client process c located at address A_c has a request to send to π, it selects a collection of network nodes Q_c and **queries** each node in Q_c for the address of π. When $P_s \cap Q_c \neq \emptyset$, the node(s) in the intersection will return a message to c stating that π is available at A_s. If $P_s = \{s\}$ and $Q_c = U$ then the technique is called **broadcasting**; if $P_s = U$ and $Q_c = \{c\}$ then the technique is called **sweeping**.

A class of distributed algorithms for match-making between client processes and server processes in computer networks is presented and the expected performance of one such algorithm using random choices is analysed. Subsequently, the optimal lower bound is determined on the performance in number of message passes for any such algorithm, in any network, under any strategy, distributed or not. This yields a combinatorial lemma which results in a lower bound on the trade-off product between the number of nodes a server advertises at and the number of nodes a client inquires at. The method will be applied to particular networks, both designed networks and spontaneously emerged networks. Finally, a probabilistic and a hashing algorithm for match-making are briefly investigated.

2.6.2. A Theory of Distributed Match-Making

In this section lower bounds are derived on the message pass complexity of a class of locate algorithms (called Shotgun Locate), for the entire range from centralised to distributed methods, and for any network topology. In the next section we give methods which achieve these lower bounds, or nearly achieve these lower bounds, for some network topologies.

The networks under consideration are point-to-point (store-and-forward) communications networks described by an undirected communications graph $G=(U,E)$, with a set of nodes U representing the processors of the network, and a set of edges E representing bidirectional noninterfering communication channels between them. No common memory is shared by the node-processors. Each node processes messages it receives from its neighbours, performs local computations on messages and sends messages to neighbours. A *message pass* or *hop* consists of the sending of a message from one node to one of its direct neighbours.

The number of message passes needed for match-making depends on the topology of a network. We want to obtain topology independent lower bounds. Therefore, assume that all messages can be routed in one message pass to their destinations. Equivalently, assume that the network is a *complete* graph. Lower bounds on the needed number of message passes in complete networks *a fortiori* hold for all networks.

For each network $G=(U,E)$ and associated match-making algorithm, there are total functions P, Q such that:

$$P,\, Q\colon U \rightarrow 2^U \quad .$$

(Here 2^U is the set of all subsets of U.) Any server residing at node i starts its stay there by *posting* its (port, address) pair at each node in $P(i)$. Any client residing at node j *queries* each node in $Q(j)$ for each service (port) it requires.

We assume that all nodes j have a **cache** which is large enough to store all (port, address) pairs associated with addresses i such that $j \in P(i)$. That is, the nodes at which the **rendez-vous'** are made can hold all posted material. The caches are large enough to hold so many (port, address) pairs that they never have to discard one for a server that is still active. Entries are made or updated whenever a message is received from a server process with its address (or when a reply from a locate operation is received). The messages can be timestamped to determine which addresses are out of date in case of a conflict.

We have dubbed this class of algorithms **Shotgun Locate** algorithms. (Put so many pheasants in the bushes that the hunter can expect success for the amount of shot he is willing to spend.) Later we consider alternative locate methods: **Hash Locate** where the functions P, Q depend on the service ports as well, and **Lighthouse Locate** which is a probabilistic version of Shotgun Locate where too-small caches can discard (port, address) pairs.

2.6.3. Probabilistic Analysis

Let the number of elements in U (universe) be n. Let a given server s reside at node i. Let p be the cardinality of $P(i) \subseteq U$, the set of nodes where s posts its whereabouts. Let a given client c reside at node j. Let q be the number of elements in $Q(j) \subseteq U$, the set of nodes queried by c. If the elements of $P(i)$ and $Q(j)$ are randomly chosen then the probability for any one element of U to be an element of $P(i)$ $[Q(j)]$ is p/n $[q/n]$. If $P(i)$ and $Q(j)$ are chosen independently then the probability for any one element of U to be an element in both $P(i)$ and $Q(j)$ is pq/n^2. Since there are n elements in U, the expected size of $P(i) \cap Q(j)$ is given by

$$E(\#(P(i) \cap Q(j))) = \frac{pq}{n} .$$

Therefore, to expect one full node in $P(i) \cap Q(j)$, we must have $p+q \geqslant 2\sqrt{n}$. This is the situation for a particular pair of nodes. For the performance of the whole network we have to consider the combined performance of the n^2 pairs of nodes. The above analysis holds for each pair i, j of elements of U, since they are all interchangeable. Consequently, the minimal *average* value of $p+q$ over all pairs in U^2 must be $2\sqrt{n}$, in order to expect successful match-making for *each* pair.

By careful and deliberate choice of the sets $P(i)$ and $Q(j)$, we may improve the situation in two ways:

- The sizes of P and Q can be reduced without reducing the probability of success.
- The probability of success can be increased without increasing the sizes of P and Q.

It is, of course, especially interesting to increase the probability of success to certainty, and to derive the minimum sizes for P and Q needed to achieve certain success. This is the subject of the next section.

2.6.4. Number of Messages for Match-Making

To match a server at node i to a client at node j the following actions have to take place. The server at i tells a set $P(i)$ of nodes about its location. Client j queries a set $Q(j)$ of nodes for the desired service. Call the set of nodes $r_{i,j} = P(i) \cap Q(j)$ the set of **rendez-vous** nodes, that is, the nodes at which a *rendez-vous* between a client at j looking for a service and a server at i offering that service can be made.

Definition. The $n \times n$ matrix, R, with entries $r_{i,j}$ $(1 \leqslant i,j \leqslant n)$ is the *rendez-vous* matrix. Each entry $r_{i,j}$, represents the set of *rendez-vous* nodes where the client at node j can find the location i and port of the server at node i. Note that:

$$\bigcup_{j=1}^{n} r_{i,j} \subseteq P(i) \quad \text{and} \quad \bigcup_{i=1}^{n} r_{i,j} \subseteq Q(j) \quad . \tag{M1}$$

To prevent waste in message passes, we can take care that the inclusions in (M1) are replaced by equalities. An optimal shotgun method has exactly *one* element in each $r_{i,j}$. Below, we represent such singleton sets by their single element. (If faults occur in the network then we may opt for more redundancy by using larger $r_{i,j}$.

2.6.5. Examples of rendez-vous matrices

1. *Broadcasting*. The server posts only to itself and the client queries every node:

		Clients								
		1	2	3	4	5	6	7	8	9
	1	1	1	1	1	1	1	1	1	1
S	2	2	2	2	2	2	2	2	2	2
e	3	3	3	3	3	3	3	3	3	3
r	4	4	4	4	4	4	4	4	4	4
v	5	5	5	5	5	5	5	5	5	5
e	6	6	6	6	6	6	6	6	6	6
r	7	7	7	7	7	7	7	7	7	7
s	8	8	8	8	8	8	8	8	8	8
	9	9	9	9	9	9	9	9	9	9

2. *Sweeping*. The client stays put and the server looks for work:

		Clients								
		1	2	3	4	5	6	7	8	9
	1	1	2	3	4	5	6	7	8	9
S	2	1	2	3	4	5	6	7	8	9
e	3	1	2	3	4	5	6	7	8	9
r	4	1	2	3	4	5	6	7	8	9
v	5	1	2	3	4	5	6	7	8	9
e	6	1	2	3	4	5	6	7	8	9
r	7	1	2	3	4	5	6	7	8	9
s	8	1	2	3	4	5	6	7	8	9
	9	1	2	3	4	5	6	7	8	9

3. *Centralised name server*.

All services post at node 3 and all clients query for services at node 3:

		Clients								
		1	2	3	4	5	6	7	8	9
	1	3	3	3	3	3	3	3	3	3
S	2	3	3	3	3	3	3	3	3	3
e	3	3	3	3	3	3	3	3	3	3
r	4	3	3	3	3	3	3	3	3	3
v	5	3	3	3	3	3	3	3	3	3
e	6	3	3	3	3	3	3	3	3	3
r	7	3	3	3	3	3	3	3	3	3
s	8	3	3	3	3	3	3	3	3	3
	9	3	3	3	3	3	3	3	3	3

4. *Truly distributed name server*.

All nodes are used equally often as *rendez-vous* node:

Clients

		1	2	3	4	5	6	7	8	9
	1	1	1	1	2	2	2	3	3	3
S	2	1	1	1	2	2	2	3	3	3
e	3	1	1	1	2	2	2	3	3	3
r	4	4	4	4	5	5	5	6	6	6
v	5	4	4	4	5	5	5	6	6	6
e	6	4	4	4	5	5	5	6	6	6
r	7	7	7	7	8	8	8	9	9	9
s	8	7	7	7	8	8	8	9	9	9
	9	7	7	7	8	8	8	9	9	9

5. *Hierarchically distributed name server.*
Links for nodes lower in the hierarchy are served by *rendez-vous* nodes higher in the hierarchy. Networks are hierarchically ordered as shown in FIGURE 2.5:

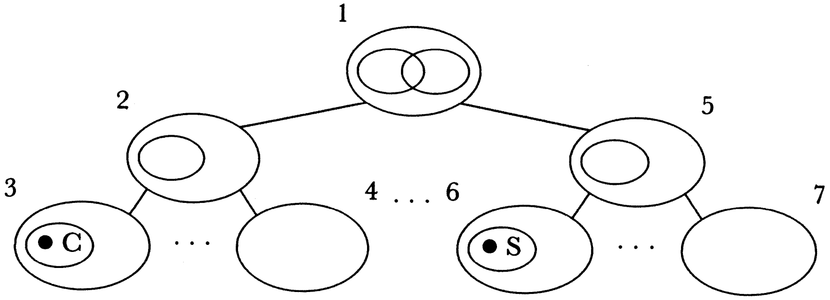


FIGURE 2.5. A hierarchy of networks; nodes higher in the network hierarchy serve as rendez-vous nodes for nodes in lower levels.

Clients

		1	2	3	4	5	6	7	8	9
	1	7	7	7	9	9	9	9	9	9
S	2	7	7	7	9	9	9	9	9	9
e	3	7	7	7	9	9	9	9	9	9
r	4	9	9	9	8	8	8	9	9	9
v	5	9	9	9	8	8	8	9	9	9
e	6	9	9	9	8	8	8	9	9	9
r	7	9	9	9	9	9	9	9	9	9
s	8	9	9	9	9	9	9	9	9	9
	9	9	9	9	9	9	9	9	9	9

6. *Distributed name server* for the binary 3-cube topology. The node addresses are the 3-bit addresses of the corners of the cube. For all $a,b,c \in \{0,1\}$, $P(abc) = \{axy \mid x,y \in \{0,1\}\}$ and $Q(abc) = \{xbc \mid x \in \{0,1\}\}$:

		Clients							
		000	001	010	011	100	101	110	111
	000	000	001	010	011	000	001	010	011
S	001	000	001	010	011	000	001	010	011
e	010	000	001	010	011	000	001	010	011
r	011	000	001	010	011	000	001	010	011
v	100	100	101	110	111	100	101	110	111
e	101	100	101	110	111	100	101	110	111
r	110	100	101	110	111	100	101	110	111
s	111	100	101	110	111	100	101	110	111

2.6.6. Lower Bound for Complete Networks

There are n possible *rendez-vous* nodes and n^2 elements in R. By choice of P and Q, the algorithm distributes the load of being a *rendez-vous* node over the nodes in the network. It is sometimes preferable to distribute the load unevenly. For instance, in the very large networks with millions of processors which are now envisioned, $\sqrt{n}$ message passes is just too much because n is so large. In hierarchical networks (Example 5) the average number of message passes for a match-making instance can be as low as $O(\log n)$. This means that some nodes are used very often as *rendez-vous* node, and others very seldom or not at all. A combination of hierarchical and local posting may also be useful.

Let the *rendez-vous* matrix R have n^2 node entries, constituted by $k_i \geqslant 0$ copies of each node i, $1 \leqslant i \leqslant n$. Clearly,

$$\sum_{i=1}^{n} k_i = n^2 \quad , \tag{M2}$$

To match a server at node i with a client at node j, the server sends messages to all nodes in $P(i)$ and the client sends messages to all nodes in $Q(j)$. So, all in all, the *number of message passes* $m(i,j)$ involved in this *match-making instance* is given, in a complete network, by

$$m(i,j) = \#P(i) + \#Q(j) \quad . \tag{M3}$$

In the examples above we have seen that, for different pairs i,j, the number of message passes $m(i,j)$ for a match-making instance can, in a single *match-making strategy*, range all the way from a minimum of 2 to n, and beyond. We determine the quality and complexity of a match-making strategy by the *minimum* of $m(i,j)$, the *maximum* of $m(i,j)$ and, above all, the *average* of $m(i,j)$, for $1 \leqslant i,j \leqslant n$.

Definition. The *average* number of message passes $m(n)$ of the given match-making strategy (which is determined by the *rendez-vous* matrix R) is:

$$m(n) = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} m(i,j) \quad . \tag{M4}$$

We shall now derive an exact lower bound on $m(n)$ expressed in terms of the number k_i of times node i occurs in R, that is, the number of times k_i is used as *rendez-vous* for a pair of nodes.

Proposition 1. *Consider the rendez-vous matrix R as defined. Then $m(n)$ is bounded below by:*

$$\frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\#P(i)\#Q(j) \geqslant \frac{1}{n^2}\left[\sum_{i=1}^{n}\sqrt{k_i}\right]^2 \tag{M5}$$

Proof. Let r_i [c_i] be the number of different nodes in row i [column i] $(1 \leqslant i \leqslant n)$. Then

$$r_i = \#\bigcup_{j=1}^{n} r_{i,j} \quad \& \quad c_j = \#\bigcup_{i=1}^{n} r_{i,j} \ . \tag{1}$$

Let R_i be the number of different rows containing node i, and let C_i be the number of different columns containing node i $(1 \leqslant i \leqslant n)$. Let $\rho_{i,j} = 1$ if node i occurs in row j, otherwise $\rho_{i,j} = 0$, and let $\gamma_{i,j} = 1$ if node i occurs in column j, otherwise $\gamma_{i,j} = 0$, $(1 \leqslant i, j \leqslant n)$. Then,

$$\sum_{j=1}^{n} r_j = \sum_{j=1}^{n}\sum_{i=1}^{n}\rho_{i,j} = \sum_{i=1}^{n} R_i \tag{2}$$

$$\sum_{j=1}^{n} c_j = \sum_{j=1}^{n}\sum_{i=1}^{n}\gamma_{i,j} = \sum_{i=1}^{n} C_i \ .$$

Clearly, for all i $(1 \leqslant i \leqslant n)$ we have

$$R_iC_i \geqslant k_i \ . \tag{3}$$

Furthermore, since

$$k_jR_i^2 - 2\sqrt{k_ik_j}\,R_iR_j + k_iR_j^2 = (\sqrt{k_j}\,R_i - \sqrt{k_i}\,R_j)^2$$
$$\geqslant 0 \ ,$$

for all i,j $(1 \leqslant i,j \leqslant n)$, we obtain immediately:

$$\frac{k_jR_i}{R_j} + \frac{k_iR_j}{R_i} \geqslant 2\sqrt{k_ik_j} \ ,$$

from which it follows that:

$$\sum_{i=1}^{n} R_i \sum_{j=1}^{n} k_jR_j^{-1} \geqslant \sum_{i=1}^{n}\sum_{j=1}^{n}\sqrt{k_ik_j} \ . \tag{4}$$

Hence,

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\#P(i)\#Q(j) \geqslant \sum_{i=1}^{n}\sum_{j=1}^{n} r_ic_j \quad \text{(by (M1) \& (1))}$$
$$= \sum_{i=1}^{n} r_i \times \sum_{j=1}^{n} c_j$$

$$= \sum_{i=1}^{n} R_i \times \sum_{j=1}^{n} C_j \quad \text{(by (2))}$$

$$\geqslant \sum_{i=1}^{n} R_i \times \sum_{j=1}^{n} \frac{k_j}{R_j} \quad \text{(by (3))}$$

$$\geqslant \left[\sum_{i=1}^{n} \sqrt{k_i} \right]^2 \quad \text{(by (4))}.$$

The Proposition follows. □

The constraints (M1) to (M5) imply a lower bound trade-off between the number of message passes for posting (and nodes for storing) a server's (port, address) and the number of message passes for querying nodes for the whereabouts of services.

The distributed match-making strategy can be adjusted to the relative frequency of these happenings, so as to minimise the weighted overall number of messages. For instance, if the average call for a service at i by a client at j occurs $\alpha_{i,j}$ times more often than the average posting of a service available at i, then we may want to minimise $m(n)$ replacing (M3) by (M3′):

$$m(i,j) = \#P(i) + \alpha_{i,j}\#Q(j) \ . \qquad \text{(M3′)}$$

Proposition 1 immediately gives a lower bound on the average number of messages involved with a *rendez-vous*:

Proposition 2. *For a complete n-node network and any Shotgun Locate strategy, with the k_i's as defined above, the average number $m(n)$ of message passes (c.q., distinct nodes accessed) to make a match is*

$$m(n) \geqslant \frac{2}{n} \sum_{i=1}^{n} \sqrt{k_i} \ .$$

Proof. Assume, by way of contradiction, that the Proposition is false, that is,

$$m(n) < \frac{2}{n} \sum_{i=1}^{n} \sqrt{k_i}$$

But

$$\frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} (r_i + c_j) \leqslant m(n) \quad ,$$

so

$$\sum_{i=1}^{n} \sum_{j=1}^{n} (r_i + c_j) = n \sum_{i=1}^{n} (r_i + c_i)$$

$$< 2n \sum_{i=1}^{n} \sqrt{k_i} \ .$$

Then,

$$\sum_{i=1}^{n} r_i \sum_{i=1}^{n} c_i < \left[\sum_{i=1}^{n} \sqrt{k_i} \right]^2 ,$$

which contradicts Proposition 1. □

Propositions 1 and 2 hold *mutatis mutandis* for nonsquare matrices R, that is, for networks where some nodes can host only servers and other nodes perhaps only clients.

In the *truly distributed case*, that is, $k_1 = k_2 = \cdots = k_n = n$, Propositions 1 and 2 specialise to the Corollary below. Here, each node occurs equally often as *rendez-vous* node in matrix R, and hence carries an equal load of the match-making work.

Corollary. *Consider the rendez-vous matrix R as defined, for $k_1 = k_2 = \cdots = k_n = n$. Then:*

$$\frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \#P(i)\#Q(j) \geqslant n ,$$

$$m(n) \geqslant 2\sqrt{n} .$$

This is the same average number of messages as in the probabilistic approach described earlier. For certain success, and perfect distribution of the match-making load, the sum of the sizes of P and Q can not be reduced below $2\sqrt{n}$, where n is the number of nodes in the network.

Another choice of the k_i's gives:

Corollary. *For $k_2 = k_3 = \cdots = k_n = 0$ and $k_1 = n^2$, that is, there is a centralised name server at node 1, we obtain:*

$$\frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \#P(i)\#Q(j) \geqslant 1 ,$$

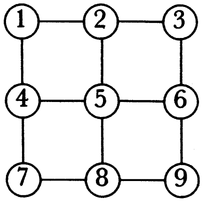
$$m(n) \geqslant 2 ,$$

which is not surprising.

2.6.7. Implementations in Particular Networks

Manhattan Networks

In a *Manhattan Network*, the network is laid out as a $p \times q$ rectangular grid of nodes. If the availability of a service is posted along the server's row and requests for a service along the client's column, caches must be of size $O(q)$ and the number of message passes for each match-making instance is $O(p+q)$. For $p = q$ we have $m(n) = 2\sqrt{n}$ and caches of size $\sqrt{n}$. For the 9-node network below,



the *rendez-vous* matrix looks as follows:

		Clients								
		1	2	3	4	5	6	7	8	9
	1	1	2	3	1	2	3	1	2	3
S	2	1	2	3	1	2	3	1	2	3
e	3	1	2	3	1	2	3	1	2	3
r	4	4	5	6	4	5	6	4	5	6
v	5	4	5	6	4	5	6	4	5	6
e	6	4	5	6	4	5	6	4	5	6
r	7	7	8	9	7	8	9	7	8	9
s	8	7	8	9	7	8	9	7	8	9
	9	7	8	9	7	8	9	7	8	9

Wrap-around versions of the method can also be used in cylindrical networks, or torus-shaped networks. It is, in fact, the method used in the torus-shaped Stony Brook Microcomputer Network.[25]

Multidimensional Cubes

Suppose the network is laid out as a d-dimensional cube. The 2^d nodes can be named by d-bit numbers in such a way that, if two nodes are connected, their names differ in a single bit. Assume that d is even.

Server's Algorithm. A server at an address $s = s_1 s_2 \cdots s_d$ broadcasts its (port, address) along a spanning tree to all nodes in the $d/2$-dimensional cube spanned by the nodes in

$$P(s) = \{a_1 a_2 ... a_{\frac{d}{2}} s_{\frac{d}{2}+1} ... s_d \,|\, a_1, ..., a_{\frac{d}{2}} \in \{0,1\}\} \ .$$

Client's Algorithm. A client at an address $c = c_1 c_2 \cdots c_d$ broadcasts its query along a spanning tree to all nodes in the $d/2$-dimensional cube spanned by the nodes in

$$Q(c) = \{c_1 c_2 ... c_{\frac{d}{2}} a_{\frac{d}{2}+1} ... a_d \,|\, a_{\frac{d}{2}+1}, ..., a_d \in \{0,1\}\} \ .$$

For each pair $s, c \in \{1, \ldots, n\}$ the *rendez-vous* node is given by

$$P(s) \bigcap Q(c) = \{c_1 c_2 ... c_{\frac{d}{2}} s_{\frac{d}{2}+1} ... s_d\} \ .$$

The number of message passes is the same for each server-client pair, and therefore

$$m(n) = \#P(s) + \#Q(c) = 2\sqrt{n} \ .$$

The nodes need $O(\sqrt{n})$-size caches.

Projective Plane Topology.

The projective plane $PG(2,k)$ has $n = k^2 + k + 1$ points and equally many lines. Each line consists of $k + 1$ points and $k + 1$ lines pass through each point. Each pair of lines has exactly one point in common. In a network with projective plane topology, a server s posts its (port, address) to all nodes on an arbitrary line incident on its host node. A client c queries all nodes on an arbitrary line incident on its own host node. The single common node of the two lines is the *rendez-vous* node. A $\sqrt{n}$ size cache for each node suffices. Since the nodes are symmetric, it is easy to see that

$$m(n) = \#P(s) + \#Q(c) = 2(k+1) \approx 2\sqrt{n} \ .$$

This combination of topology and algorithm is resistant to *failures* of lines, provided no point has all lines passing through it removed.

Hierarchical Networks

Local-area networks are often connected, by *gateway* nodes, to wide-area networks, which, in turn, may also be interconnected. Locating services and objects in such network hierarchies is bound to become an acute problem.

Service naming preferably should be resolved in a way which is machine-independent and network-address-independent. Consequently, ways will have to be found to locate services in very large networks of hierarchical structure. There, the truly distributed $\sqrt{n}$ solutions to the locate problem are not acceptable any more. Fortunately, in network hierarchies, it can be expected that local traffic is most frequent: most message passing between communicating entities is intra-host communication; of the remaining inter-host communication, most will be confined to a local-area network, and so on, up the network hierarchy. For locate algorithms these statistics for the locality of communication can be used to advantage. When a client initiates a locate operation, the system first does a local locate at the lowest level of the network hierarchy (e.g., inside the client host). If this fails, a locate is carried out at the next level of the hierarchy, and this goes on until the top level is reached.

Assume that a level i network connects n_i level $i-1$ networks through n_i gateways, for each $1 < i \leqslant k$ (or basic nodes, at the lowest level 0 for $i = 1$). Assume also that the n_i gateway hosts compose a level i network with a topology which allows thrifty truly distributed match-making with $2\sqrt{n}$ message passes per match, for all $i \geqslant 1$.

Server's Algorithm. A server posts its (port, address) by selecting $\sqrt{n_i}$ gateways, connecting level $i-1$ level networks in a level i network, at each level i of the hierarchy, on a path from its host node to the highest level network, to advertise their location.

Client's Algorithm. Similarly, at each level i on a path from its host node to the highest level network, a client's locate in a network of that level can be done in $O(\sqrt{n_i})$ message passes.

This gives an average message pass complexity

$$m(n) \in O(\sum_{i=1}^{k} \sqrt{n_i})$$

for a hierarchical network with a total of $n \leqslant \prod_{i=1}^{k} n_i$ nodes. Assuming that all n_i's equal a fixed α, the number of levels in the hierarchy is k, and the total number of nodes in the network is $n = \alpha^k$ then the message pass complexity of the locate is $m(n) \in O(k\sqrt{\alpha})$. Therefore,

$$m(n) \in O(kn^{\frac{1}{2k}}) \ .$$

Having the number k of levels in the hierarchy depend on n, the minimum value

$$m(n) \in O(\log n)$$

is reached for $k = ½\log n$. This message pass complexity is much better than $\Omega(\sqrt{n})$, but the cache size and the match-making load towards the top of the hierarchy increase rapidly. Essentially, the cache of a node has to hold as many (port, address)'s as there are nodes in the subtree it dominates. In some cases this can be avoided. For in a network hierarchy, as we have sketched, services are often exclusively accessed by local clients. A service that 'knows' it is local can do its posting local. This is, in fact, what the ***Amoeba*** *local* call does, described in § 2.4.1.

2.6.8. Lighthouse Locate

We imagine each grid point in the 2-dimensional Euclidean plane to hold a processor. Communication between processors takes place over the grid lines. The number of servers satisfying a particular port in an n-element region of the grid has expected value sn for some fixed constant $s > 0$.

In this (infinite) network, there are $2h^2 + 2h + 1$ nodes within a radius of h hops, so the expected number of servers in a radius of $\sqrt{1/2s}$ is 1.

The *Lighthouse Locate* algorithm uses this in the following way: A server advertises its presence by posting its *(port, address)* to a random node at a distance of

$$l = \left\lceil c\sqrt{\frac{1}{2s}} \right\rceil$$

hops, where c is a constant. This message passes $l - 1$ nodes on its way to its destination. Each node on the path plus the destination node store the server's (*port*, *address*) in their cache. This is repeated at regular intervals; the server chooses another node each time.

Simultaneously, nodes on previously visited paths discard their information as their caches fill up with information from other servers. The net effect is, that a server advertises its presence like a lighthouse: it sends out beams that leave a trail for a short time and then extinguish. This leads to the following algorithm:

Server's Algorithm. Each server sends out a random direction beam of length l every δ time units. Each trail left by such a beam disappears after d time units. That is, a node discards a (port, address) posting after d time units. Assume that the time for a message to run through a path of length l is so small in relation to d that the trail appears and disappears instantaneously.

Client's Algorithm. To locate a server, the client beams a request in a random direction at regular intervals. Originally, the length of the beam is l' and the intervals are δ', where l' and δ' are constants of the algorithm. After e unsuccessful trials, the client increases its effort by doubling the length of the inquiry beam and the intervals between them ($l' \leftarrow 2l'$ & $\delta \leftarrow 2\delta$). And so on.

Another possibility is to let the length of the locate beam be $2^{t_i} l'$, where t_i, ($i = 1, 2, \ldots$) is

$$010201030102010401020103010201050102010 3 \cdots$$

Here the length of the locate beam is $2^i l'$ once in each interval of 2^{i+1} trials. (This sequence is sequence 51 in Sloane's catalogue,[66] sometimes referred to as the *binary carry sequence*.) The schedule can conveniently be maintained by a binary counter: the position i of the most significant bit changed by the current unit increment indicates the current beam length $2^i l'$. This schedule has the additional advantage that the servers which drift nearer to the client are located more quickly.

Before the locate method for the euclidean plane can be converted into a practical algorithm for locating services it is necessary to find ways of mapping point-to-point networks onto the euclidean plane in such a way that the euclidean plane algorithm can be converted into an algorithm for a point-to-point network. Fortunately, such a mapping can often be found. Most point-to-point networks have routing tables that tell each node which outgoing arc to use to get a message to its destination. In [12] these tables are used back-to-front to broadcast messages over the network in near optimal fashion. We can use these tables back-to-front to simulate sending messages along "a straight line" of certain length. The technique is as follows.

A client (or server) wishing to send a beam of length k (using message passes as the unit of length) chooses a random outgoing arc and sends the message along it to its neighbour. This neighbour, upon reception of such a message decreases the hop count (in the message) by one, and sends the message on any one outgoing arc that is used to send messages *from* the node at the *other end* of the arc *to* the *original* client (or server) where the beam started from. And so on, until the hop count reaches 0.

2.6.9. Hash Locate

Let in a given network $G=(U,E)$ the set of ports (i.e., types of services available) be Π. We can define the functions P and Q as in Shotgun Locate, this time using the port identities as well:

$$P,Q\colon U\times\Pi\rightarrow 2^U \ .$$

If we are dealing with a very large network, where it is advantageous to have servers and clients look for nearby matches, we can hash a service onto nodes in neighbourhoods. A neighbourhood can be a local network, but also the network connecting the local networks, and so on. Therefore, such functions can be used to implement the idea of certain services being local and others being more global (*cf.* the section on hierarchically structured networks) thus balancing the processing load more evenly over the hosts at each level of the network hierarchy. Like Shotgun Locate, the Hash Locate below is a specialisation of this more general method.

In **Hash Locate** we construct hash functions that map service names onto network addresses. That is,

$$P,\ Q\colon \Pi\rightarrow 2^U \quad \& \quad P=Q.$$

This technique is very efficient. Each server s posts its (port, address) at the node(s) $P(\pi)$, if π is the port of s, and each client in need for a service at port π queries (one of) the node(s) in $P(\pi)$. Apart from redundancy for fault-tolerance, clients and servers need only use one network node each in every match-making. (Clearly, the *rendez-vous* matrix must be interpreted differently in this setting.) Provided the hash function is well-chosen, it distributes the burden of the locate work over the network. It suffers from the drawback that, if nodes are added to the network, the hash function must be changed to incorporate these nodes in the set of potential *rendez-vous* nodes.

3

SERVICES, OBJECTS AND CAPABILITIES

A **service** is an abstraction. It allows its users to access and manipulate **objects** without knowledge of the implementation of the service or the structure of the objects. In this sense, objects and services can be compared to *abstract data types*. An abstract data type can only be accessed through a well defined set of operations; an ***Amoeba*** object can only be accessed through the service that implements the object. The set of operations on an ***Amoeba*** object is defined by the service that implements the object.

The difference between abstract data types and services lies in the way they are used. An abstract data type is a way of structured programming; even when sharing the same abstract data types, programs do not share the physical objects created as a consequence of using the abstract data types. With services this is different: two processes, using the same service often share the same physical *server process*, and additional mechanisms are needed to prevent users from accidently, or maliciously manipulating each other's objects.

This chapter presents general mechanisms for accessing services and the objects managed by them using a *capability mechanism*. This mechanism, which is integrated with the capability-based addressing mechanisms of the previous chapter, allows a client to operate on objects, managed by a service using *requests* and *replies*, with semantics defined by the designer of the service.

Requests and replies come in pairs. A client process sends a request to a server process, the server carries out the request and returns a reply. These message pairs, **transactions**, are supported by the **transaction protocol** which is used to reliably communicate requests and replies between clients and servers in as few messages as possible.

The remainder of this chapter is structured as follows: § 3.1 discusses object oriented protection with capabilities; § 3.2 contains a description of the Transaction Layer, the means by which clients and servers interact. § 3.3 describes the **Directory Service**, a service that provides storage and retrieval of capabilities, and discusses means to protect capabilities from theft.

3.1. Capabilities

The ***Amoeba*** capability system allows *object oriented protection*: A client process requests a service to create an object of a certain type and receives a capability for the newly created object. This capability allows the client to address the object in future requests, and prevents other processes that do not possess the capability for that particular object from accessing it.

In the previous chapter it was shown that ports are capabilities for communication between processes, and that these capabilities are handled directly by the processes holding them. The same applies to capabilities for objects. This contrasts sharply with the management of capabilities in other capability systems. Usually, capabilities are maintained by the operating system, where users can only manipulate them through system calls. Examples of such systems are Hydra,[40, 28] and Roscoe.[67]

In ***Amoeba***, capabilities are not managed by the operating system, but they are stored directly in user space. There are several reasons for this. To begin with, the ***Amoeba*** operating system kernels need not be secure, so they are not very useful as reliable managers of capabilities. Furthermore, it will be shown that there is no necessity for keeping capabilities out of the user's reach, because, properly constructed, capabilities can not be misused, even though they are kept in user space. ***Amoeba*** keeps capabilities in user space, and it will be shown that the ***Amoeba*** capability system is just as powerful as other capability systems where a secure operating system safekeeps capabilities and carries out operations on them.

A capability is a ticket the possession of which allows the holder to carry out certain operations on a specific object. To carry out the same operations on another object requires a different capability; other operations on the same object also require another capability. The operations allowed by a capability on an object by its holder are usually called **rights**. A capability can hold many rights, allowing its holder to carry out many operations on the object it is for, or it can have just one right, allowing its holder just to carry out the operations allowed by the one right.

To illustrate the use of these rights, consider a file system. The objects of the file system are files. When a file is created, the owner usually receives a capability for the file with all the rights on. Files are often shared, however, and typically the owner would like to give other users permission to read, but not write the file, or permission to read and write, but not delete the file, or permission to read and append to the file, but not destroy any information already on the file. The rights for a file capability are typically: *owner*, which allows the holder to remove the file, create new capabilities with other rights, or invalidate all other capabilities; *read*, which allows the holder to read the file, but nothing else; *write*, which allows the holder to write the file, but not read it; and *append*, which allows the holder to append to the file, but not overwrite existing information.

When a service is designed, the operations on its objects are defined, and what rights are needed to allow holders of capabilities to carry them out. For each type of object there is a different set of operations and a different set of rights.

3.1.1. Capabilities and Ports

Capabilities are closely related to ports. To get access to an object, managed by some server, a client process needs a capability to access the service, and then a capability to access the object within the service. We can think of the concatenation of these two capabilities as a capability for the object, and view the service as an abstract data type, defining the set of operations on the object.

An ***Amoeba*** capability for object O managed by service S is defined as a bit string, made up of two parts, the *put-port* for service S, and a *private part*, interpreted by service S as a capability for object O. This is illustrated in FIGURE 3.1.

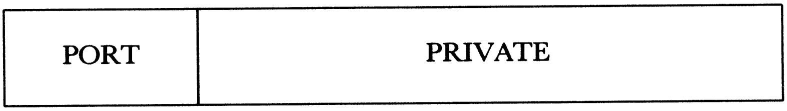


FIGURE 3.1. Layout of a capability with a port-part and a private part

The designer of the service is free to determine both the length and the semantics of the private part. Most services to date use an 80-bit private part for a 128-bit capability. Often, the private part will contain an *object* field which is used as an index into a table of objects, a *rights* field, indicating what rights the capability holds for the object, and a *check* field which makes the capability sparse (see FIGURE 3.2). Sparseness is essential for protection; capabilities are kept in user space, so a malicious user may try to forge capabilities by creating bit strings and passing them off as capabilities. By making capabilities sparse enough, the probability of a user succeeding in coming up with a bit string that is a legal capability can be made arbitrarily small.

The *check* field of a capability must depend on the *object* field and the *rights* field. If this were not the case it would be trivial, knowing a capability for one object, to create capabilities with different rights for that object, or to construct capabilities for other objects as well. Typically a server will encrypt the object number, the rights field and possibly a random number, using a one-way function* and use the result as the *check* part of the capability.

* One-way functions were discussed in § 2.3.3.

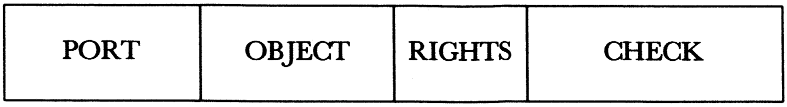


FIGURE 3.2. Layout of a typical capability

Servers are not required to adhere to the capability structure of FIGURE 3.2, but it is a general form, and it will be used to illustrate how operations on capabilities can be implemented. Some services, for instance, may not distinguish different rights on their objects, so they will not have a *rights* field. Other services may not use objects (*e.g.*, compiler service), but have different rights (*e.g.*, for debugging options); such services will not use an *object* field.

3.1.2. Operations on Capabilities

The operations on capabilities are not just limited to presenting them to services to carry out operations on objects. It is possible to give capabilities away to other clients, giving them permission to perform the same operations on the object. New capabilities can be derived from old ones with a subset of the rights of the original one, or capabilities can be made invalid. In this section the operations on capabilities are discussed: *capability passing*, *restriction*, and *retraction*.

Capability passing is trivial in ***Amoeba***. Capabilities are stored directly in user space, so users have direct control over them. A user with a capability for an object can give another user the same rights to the object by just giving away a copy of the capability. In fact, there is no way to prevent users from giving away capabilities.[19] By some, this is viewed as a defect of our capability system. Suppose user A holds a capability for some object, and A is not allowed to give that capability to user B. However, A can give a *capability for that capability* to B: All A needs to do is forward requests for operations using that capability from B to the server. A has then effectively given away the capability to B. It is because of this that we have decided it is useless to restrict capability passing in ***Amoeba***.

Although we believe it is useless to restrict capability passing, it is not difficult to implement in ***Amoeba***. There is a simple mechanism which makes a capability that works in the hands of one user useless in the hands of another. In order to make this work, users must provide an authenticating *signature* with each request (see § 2.3.2). A service that wants to hand out a capability that works in the hands of just one user, encrypts the original capability with the client's ciphertext signature before giving it to that user.[*] When the capability is subsequently presented to the service, the original is encrypted with the received ciphertext signature and compared to the presented capability. If there is a match, the capability was valid. If not, either the capability or the signature was wrong; in the latter case, it was a capability that was given away.

In this scheme it is still possible to give capabilities away to other users, but this requires giving away the authenticating signature as well. We do not know any systems, however, where this is not possible: Most operating systems that claim to be secure, use passwords for user authentication; giving away one's signature in ***Amoeba*** amounts to the same thing as giving away one's password in another operating system.

Before giving away a capability to another user, it is often necessary to *restrict* that capability; that is, take away some of the rights in the capability. Again we consider the example of a file server. When a client requests the file server to create a file, the server returns an *owner* capability, a capability with all the rights on. Often the client will allow others to read the file, and perhaps allow some to write it or append to it as well. It will be seldom, however, that a client will give the *owner* right away, the right to remove the file altogether, and the right to retract the existing capabilities (make the existing capabilities illegal). First these rights have to be removed from the capability.

In conventional capability systems, where the capabilities are managed by the operating system this is easy: when a 'pass capability' system call is made, the client specifies which rights should be passed, and the operating system will strip some rights bits before copying the capability. In our system this is more difficult, since capabilities are in user space. If the client could just strip some rights bits and then give the capability away, there is nothing to prevent the recipient to add these bits again.

The usual way to do *restrict* is to ask the server to create a restricted capability. The capability of FIGURE 3.2, for instance, can be constructed by making the *check* field depend on the *object* and *rights* fields. To make a restricted capability, the server would clear selected *rights* bits and recompute the check field. This is a straightforward mechanism, but it requires an extra transaction with the server, each time a (new) restricted capability is required. In practise, the message traffic need not increase at all, however, if servers allow their clients to ask for a number of capabilities with different rights the moment an object is created. Apart from the *owner* capability, a user would then receive a few other capabilities 'for giving away.'

An alternative scheme exists that allows clients to construct capabilities with restricted rights themselves. Naturally, this mechanism can only be used to remove rights, not add them. The method is based on *one-way functions*, which, as it turns out, find many applications in ***Amoeba***. The mechanism works as follows.

We start with the familiar capability layout of FIGURE 3.2. We assume there are k rights, numbered from 1 to k. Each right is represented by one bit in the

* Note, that only the holder of the plaintext signature can send messages such that the receiver gets the associated ciphertext signature.

rights field, a one when the right is present, a zero when it is absent. Initially, the server constructs a capability with all the rights on, the *rights* field is set to all ones, and the *check* field is set to some random number. One random number is chosen per object, and stored along with the object.

Let us now assume there are k one-way functions, $G_1, \cdots, G_k$, one for each right, that are *commutative*, that is,

$$G_i \circ G_j(R) = G_j \circ G_i(R) \quad \text{for all functions } G_i \text{ and } G_j.$$

The functions G_i form a mapping of the *check* field onto itself. The server makes the functions G_i public, and clients can restrict capabilities by proceeding as follows: Given a capability with right i on, a new capability with right i off is created by clearing the i'th *rights* bit, and applying G_i to the *check* field. The capability thus constructed can be further restricting by repeating the procedure over and over. The commutativity of the one-way functions ensures that the order in which the rights are removed is irrelevant. It is thus used only for convenience; one could also tag a list onto capabilities providing the order of the G's onto the capability.

When presented with a capability, the server can easily check the validity by applying the one-way functions associated with the absent rights (indicated by the rights bits in the *rights* field of the presented capability) to the original random number (which was stored along with the object itself). If this yields the same *check* field as the one in the presented capability, the capability is genuine.

It is an interesting exercise to find a set of commutative one-way functions. We have succeeded in creating such a set, based on the RSA public key algorithm.[58] It is not always practical, however, due to the enormous size of the *check* field necessary for this method. This is how it works.

Choose a number n of sufficient size that it cannot be factored (this may be done by taking the product of two or more large primes, as in the RSA algorithm). Also choose k numbers $e_1, \cdots, e_k$. These numbers must be chosen in a special way, which will be explained later. Define $G_i(R)$ $(i = 1, \cdots, k)$ as follows:

$$G_i(R) \equiv R^{e_i} \bmod n$$

Then the functions G_i are commutative:

$$G_iG_j(R) \equiv R^{e_j^{e_i}} \bmod n \equiv R^{e_ie_j} \equiv R^{e_i^{e_j}} \bmod n \equiv G_jG_i(R)$$

The G_i commute, but are they one-way functions? They are, if the e_i are chosen to be relatively prime to $\phi(n)$.* In order to see this, note that each of the G_i can be viewed as an RSA public-key encryption function with the decryption function thrown away. The arguments for the robustness of the RSA algorithm also apply to our one-way functions. In Rivest, Shamir and Adleman's paper a discussion can be found on the security of the algorithm.[58]

* Euler's Totient function

The e_i must be chosen carefully. Suppose e_i is a multiple of e_j. Then, having a capability without right j, it is possible to change it into a capability without right i, by raising the *check* field to the power i / j, thus giving a capability without right i, but *with* right j. But if all the e_i are co-prime such constructions are not possible. In fact, another interesting construction becomes possible if the e_i are co-prime. Suppose a client holds two capabilities that differ in one aspect: one has right j but does not have right i and the other has right i but does not have right j. The client that holds these capabilities can combine the two to make one capability that has both right i and right j by choosing two numbers d_i and d_j, such that $e_i d_i - e_j d_j = 1$. Note that such numbers exist since e_i and e_j are co-prime. The capability that contains right i, but not right j contains some random number R, raised to the power e_j, while the other capability contains the same number R, raised to the power e_i. What is needed for the combined capability is the number R itself. This number can easily be found by computing

$$R \equiv \frac{R^{e_i d_i}}{R^{e_j d_j}} \mod n$$

This computations is feasible if $R^{e_i e_j}$ is co-prime with n. This is almost certainly the case, since n is the product of two very large primes.

It is unfortunate that this method for restricting capabilities is not practical at present for ***Amoeba*** capabilities, because of the large *check* fields needed, caused by the large moduli required for sufficient security. The method might be applied in environments where capabilities must be made very secure and their size is not too large a price to pay for that security. A possible future VLSI implementation could make it acceptable, however.

A compromise between *restrict* implemented inside the server, and *restrict* by the client using commutative one-way functions, is to use one-way functions that do not commute. These can be used in two ways: the simplest way is to allow rights to be stripped only in a predefined order. The commutative property is not necessary then. In many cases this restriction on the order in which rights can be stripped is not so severe as it seems. On a file, for instance, a typical order of the rights could be *owner*, *write*, *append*, and *read*. Only in few applications the rights will have to be taken off in any different order. In these cases, a requests could be sent to the server to create a special capability.

When rights can only be taken off in one predefined order, there is no need for a different one-way function for each right. One function will do, the number of times it is applied then serves as the indication of what rights are still present. In fact, the *rights* field can be totally dispensed with. When the size of the *check* field is taken to be the same size as that of a *port*, the standard one-way function F can be used.

A method that does not require commutative one-way functions, but allows taking away rights in any order, requires that capabilities contain the order in which rights were taken away. The server can then perform the same computations as the client in order to check the validity of a capability. This method does need k different one-way functions for k different rights. The number of orders in which k rights can be taken away is $k!$, so the number of extra bits needed to store this information is $\log_2 k!$. For example, for 4 rights, 8 bits are needed, for 8 rights, 20 bits and for 16 rights, 50 bits.

The *retract* operation invalidates all extant capabilities for an object, and gives the owner a new one. In conventional capability systems *retract* is a complicated operation, requiring indirect objects, back pointers and other complicated machinery (see, for example, [28]); in ***Amoeba*** *retract* is simple: the server need only change the object's random number, and give the owner the new capability derived from it. All extant capabilities automatically become invalid. The *retract* right is, of course, a powerful right, which should not be given away lightly. Possession of this right allows the holder to make the object inaccessible to everyone else. In practise, *retract* will be protected by the *owner* right, or by a separate rights bit.

3.2. The Transaction Layer

The paradigm of interprocess communication in ***Amoeba*** is the **transaction.** In a transaction, a client process transmits a *request* for an operation on an object to one of the server processes for the object. The server process carries out the request and returns a *reply*. The request-reply pair forms a transaction. Processes dynamically take on the role of client or server. Server processes passively wait for requests to be sent to them. Client processes actively send requests. A process can be a client and a server simultaneously.

The Port Layer, which has been described in the previous chapter, provides a datagram service between processes. The Port Layer makes an effort to deliver every message, but occasionally messages may be lost. A layer on top of the Port Layer is needed to implement reliable transactions. This layer is the **Transaction Layer**. Basically, the Transaction Layer provides calls for sending and receiving **requests** and **replies**.

In principle, the Transaction Layer can reside both in user space, and in the operating system. In the future, it could even be designed as a peripheral device, much like the F-box of the previous chapter. In this section, however, it is assumed that the Transaction Layer is in the operating system, where it is accessed by user programs via system calls.

3.2.1. The Nature of Interprocess Communication

Choosing a transaction mechanism as the paradigm for communication is not at all a natural choice. Most computer networks use virtual circuits to implement a reliable, flow-controlled byte stream between processes. The ***Amoeba*** distributed operating system has a transaction protocol, which provides communication in the form of requests and replies, messages of finite maximum length.

There is no flow control and no sequencing: a server, prepared for one request at a time, may be bombarded by many requests from many different clients all at once; two requests sent to the same server one after the other need not arrive in the same order. If a client wants flow control, it must wait for a reply before sending the next request. Compared to the smooth, error free, flow controlled byte stream of other distributed systems, the ***Amoeba*** protocols appear primitive. But the simplicity of the ***Amoeba*** Transaction Protocol is its strength.

Virtual circuits, with their flow control, sequencing and error recovery, must be set up before the first message can be transmitted, and taken down after the last message has been received. Usually, the protocols that maintain virtual circuits use complicated protocols for optimum exploitation of the available bandwidth. Often, messages are accepted out of order, and reassembled in the proper sequence after arrival. Clever piggybacking allows acknowledgements to be included in reverse traffic. But all this cleverness is expensive: Experience with virtual circuit protocols has shown that the overhead of maintaining the protocol costs so much, that the available bandwidth is not used more optimally and that the bandwidth of the transmission medium itself is often several orders of magnitude higher than the bandwidth of the protocol itself. This inefficiency of virtual circuits is one of the reasons that ***Amoeba*** uses a different philosophy.

Virtual circuits can be efficient for bulky transmissions of large bodies of data from one process to another. In contrast to virtual circuits, transactions require no set up (this will be shown later), hence transactions are efficient for short interactions between processes.

The above arguments show that if a choice is necessary between virtual circuit service and transaction service, it is important to know how many conversations between processes are of the long and bulky kind, and how many are exchanges of single messages. We believe the latter kind is far more common than the former. The reasons follow.

Traditionally, data processing consisted of sequential processing of large files. A large file would be updated once in a while, by collecting the changes, sorting them, and then updating the file while copying it. This method is rapidly becoming obsolete. By and large, computer systems will mostly be used interactively. Large (sorted) files disappear in favour of databases which provide more possibilities and can be updated in place. Techniques are being developed in the data base world to minimise network traffic by optimum allocation of the data and judiciously choosing the locations where queries are resolved. Modern interactive data base systems mostly use short transactions, large bodies of data need seldom be transferred.

Not all computer data are stored in data bases, however. There is still need for file systems, occasionally containing very large files. In text processing, for instance, files of more than a megabyte are not uncommon. File transfer is usually considered as a typical example of an operation where a virtual circuit is better suited than a transaction protocol. However, there are several reasons why transaction protocols merit investigation, even for applications such as file transfer.

First, there is the semantic argument: Interprocess Communication is most frequently used to communicate requests over the network. Even in large file transfers, there is usually a command, accompanying the file, telling the remote machine what to do with the received file. IPC semantics based on procedure call are more suited to this kind of communication than semantics based on sequential file I/O. The *Remote Procedure Call*[51] mechanism has become a popular semantic model for IPC. In this mechanism, which requires a hook into the programming language used, one process, the caller, can call a subroutine in another process, the callee. When a remote procedure call is made, a stub in the caller's address space is called, which wraps the procedure's parameters in a message and sends the message to a stub in the callee's address space. The callee stub unwraps the message, and calls the subroutine with the appropriate parameters. The procedure's return value is returned to the caller's stub in another message. The caller stub then returns the result to the calling program. The differences betwween local and remote procedure calls are hidden as much as possible. A few differences cannot be hidden, however, mostly related with passing pointers to objects in the caller's or callee's address space.

The differences between transactions and remote procedure calls are mostly syntactical: In remote procedure call, the caller calls a remote procedure, and hidden mechanisms do the message exchange. In a transaction mechanism, the client explicitly sends a request and waits for the reply; the message exchange is not hidden. But the most important difference is the following: When a remote procedure call mechanism is used, a typical client writes a piece of a file by calling the remote write procedure, *e.g.*:

result = *write*(*filename*, *data*);

If there are several file serveers in the system, the client stub must find out which one to use. In order to do this, the stub must examine the *filename* argument. Worse still, adding another file server may require changing all the stubs to incorporate knowledge about the new server.

In object-oriented systems, procedure name and object argument (e.g., *filename*) *object* with the operation (*e.g.*, write) as a parameter. Translated in a language construct it might look like

result = *filename*(*write*, *data*);

The transaction mechanism does this: a request is sent for an operation on an object; the object's capability can be viewed as the message's destination (the system routes the message to the object's server, the server then operates on the object), the operation code (*e.g.*, *write*) is a parameter of the request.

Second, a transaction protocol (which can also be used for implementing a remote procedure mechanism) is much simpler than a byte stream protocol. A byte stream protocol usually employs a sliding window mechanism and sequence numbers for sequencing messages, and it has to maintain a *connection record* for the duration of the conversation. A server, communicating simultaneously with hundreds of clients thus needs hundreds of connection records (and as many buffers for receiving messages). A transaction protocol can be kept much simpler. There is no need for a *transaction record* between transactions, so a server, while serving hundreds of clients, needs only transaction records (and buffers) for the number of simultaneously active transactions. This number can be much less and it is under control of the server.

Third, a byte stream protocol crashes when one of the communicating processes crashes. Every process, client or server, needs code that will help it to deal with a crashed partner. This leads to a duplication of effort: there is recovery code in the byte stream protocol for lost and garbled messages and recovery code in the user programs to deal with crashed communication lines and crashed processes. A transaction protocol needs no special provisions: the same mechanism is used to deal with bad communication lines, lost messages, and crashed processes. When a client process detects a failed transaction, it can usually redo the transaction immediately, to have it carried out by another server process. Services are designed to do this correctly.

Fourth, byte stream protocols are usually complicated to deal efficiently with lost and garbled messages. But local networks are very fast and extremely reliable nowadays, so the extra overhead due to the more complicated protocols probably outweighs the additional efficiency of better dealing with lost messages. Transaction protocols are simpler, so the overhead of sending messages is less.

Fifth, although byte stream protocols may be more efficient in number of messages for transfers of many messages, file transfer is often better off if a transaction protocol is used. The size of file transfers, in many cases, is too small to make a byte stream protocol efficient. As an example of the typical sizes of the files in a computer system, we have examined the files of our own Computer Science Department UNIX system. Our system is mainly used for research, program development and text processing, and measurements showed that nearly 70% of the 20,000 or so files are less than 2K bytes in size (see FIGURE 3.3). Since popular local-area network interfaces typically support packet sizes betwween 1K and 2K bytes,* this means that between half and two thirds of the files fit in one packet.

The sizes of files give a good indication of the amounts of data to be shipped through the network, because it is the natural place to store that data. Typically, a number of processes will process data that originates from a file, is passed around, modified, and ends up on another file. Many files are not read sequentially, which results in smaller chunks of data on the network. The numbers, presented above, therefore give a good indication of the maximum sizes of data transfers, while on average the transfers will be much smaller.

We are not the first to observe that the nature of most conversations in a computer system has a transaction nature; Per Brinch-Hansen[6] used the concept of *message* and *answer* in the IPC mechanism of the RC 4000 system, and the *Thoth*

* *e.g.*, 1500 bytes for Ethernet; 2044 bytes for Pronet token ring

File length	*percent*	*File length*	*percent*
0	1.12	2047	67.06
1	1.12	4095	78.57
3	1.21	8191	87.90
7	1.42	16383	94.11
15	2.18	32767	97.41
31	3.68	65535	99.01
63	6.36	131071	99.72
127	11.48	262143	99.93
255	19.82	524287	99.97
511	33.61	1048575	99.99
1023	51.86	2097151	100.00

FIGURE 3.3. Percentage of files smaller or equal to the indicated length (in bytes). This data was obtained from 600 Mbytes of files in the computers in the Computer Science Dept. of the Vrije Universiteit.

or *V* system uses similar primitives in *.Send* and *.Reply*.[8,9] The RIG system of the University of Rochester is an example of a system that has both *atomic transactions* and *connections*, the first for request and reply, the second for longer conversations.[3] These exist as two separate mechanisms however.

As an aside before getting into the technical details of the ***Amoeba*** Transaction Protocol, it is worth noting that many applications are more transaction oriented than is often realised. Consider, for example, file transfer. The traditional mechanism is to set up a connection between two processes and send the file over it. In fact, the programs carrying out the file transfer invariably consist of a main loop with *read* and *write* statements inside the loop. We maintain that since the file really is transferred in a series of discrete transactions, the most logical way of viewing the transfer is to see the reader as a client and the writer as a server. The client sends a request saying *'Give me a piece of the file'* and the server replies with the requested data. The sequence is repeated until the file is transferred.

The strongest argument for a transaction-oriented transport protocol, however, is its speed. Few local networks, using virtual circuits obtain a user process to user process throughput of more than 25 Kbytes/sec, although the network itself (*e.g.*, CSMA/CD or a ring) can transfer more than 1 Mbyte/sec. The implementation of the ***Amoeba*** transaction protocols has resulted in a 300 Kbyte per second continuous transmission rate between user processes on different machines on an otherwise idle token ring.[47]

3.2.2. Transactions

From the viewpoint of clients and servers, transactions consist of a request, sent by the client to the server, followed by a reply form the server to the client. The Transaction Layer puts requests and replies in messages and transmits them.

The Transaction Layer has to meet two goals: It must be reliable, and it must be fast. Distributed systems rely heavily on interprocess communication facilities. In fact, where system calls were used traditionally, distributed systems often use interprocess communication facilities to get the requested work carried out by a remote service process. Clearly, the efficiency of the interprocess communication mechanisms will determine the practicality of a distributed system to a great extent.

To obtain maximum efficiency, the implementation of the Transaction Layer can be dependent on the efficiency and reliability of the underlying network. Transactions are used for all interprocess communication in the system, not only for communication between processes residing on different hosts, but also between processes on the same host and even between a process and the operating system kernel. To most processes, the Transaction Layer calls are the only available system calls.

The Transaction Layer protocol for 'intra-host' communication can be totally different from the protocol for communication through the network, although, of course, the syntax and semantics of the calls must be identical. Intra-host protocols will be discussed in Chapter 4. In this section we concentrate on the Transaction Layer protocols for communication, using the facilities provided by the Port Layer.

Besides the *request* and *reply messages* the Transaction Layer employs *control messages* to make the protocol work properly. In designing the Transaction Protocol we have made an effort to make the protocol work as fast as possible, partly by minimising the number of messages transmitted in a transaction. The minimum number of messages is, of course, two: a *request message* and a *reply message*. In many transactions only these two messages are needed.

The Transaction Layer uses timers to detect various failures. *Retransmission timers* detect lost messages or crashed partner processes, *crash timers* are used to detect server crashes, and *piggyback timers* can be used to delay some *control messages* to enable them to be included in *request* or *reply messages*.

Most transactions are short. Servers can often provide a reply to a request almost immediately. Short transactions are handled most efficiently by the Transaction Protocol. In a single short transaction, the client sends a request, the server sends a reply, and the client sends a control message to acknowledge receipt of the reply. Sequences of short transactions to the same server (*e.g.*, a sequence of database queries) are handled even more efficiently: the next request serves as an acknowledgement for the reply to the previous request.

Some transactions take a long time. The Transaction Protocol allows arbitrarily long transactions. These are useful, for instance, for transactions with terminal servers and slow or elaborate services. The Transaction Protocol uses control messages at regular intervals to check if the (slow) server has not crashed. Clients also have the option of *interrupting* a transaction if it takes too long.

3.2.3. The Transaction Interface

Requests and replies have two data areas: a *data buffer* and a *parameter area*. The data buffer contains the main body of data in a request or reply, while the parameter area can be used for additional information, such as capabilities, request types, parameters to a request, information about success or failure of a transaction, etc.

These two buffers have been kept separate to reduce the need of in-core copying: Usually, the main body of data to be transmitted in a request or reply is available in the sending process' main memory. If requests have to be sent by passing a consecutive area of memory containing the request type, parameters, and the data to the Transaction Layer, copying is often unavoidable. If, however, sending a request can be done by passing two pointers, one to a small buffer with request parameters, and one to the main body of data, large amounts of in-core copying can be saved.

In-core copying is an expensive operation compared to the the speed of fast local networks. On a small cpu, a copy loop to or from user space may require as much as 10 μsec per word. For a 1000 byte message, which has to be copied twice (once at the transmitter's end, and once at the receiver's end), this costs 10 msec, while the network hardware is usually capable of transmitting such a message in less than 1 msec (after copying the message from main memory to an internal buffer using DMA). Many IPC mechanisms consist of a number of layers of protocol, each layer copying data from buffers supplied by higher layers into messages intended for lower layers and vice versa. In the design of the ***Amoeba*** Transaction Protocol, avoidance of message copying was an important consideration.

Requests and replies may also contain a *signature*. Signatures can be used for authentication purposes, since they are encrypted with the same one-way function that is used for ports. The method is described in § 2.3.2.

The Transaction Layer calls for clients are shown in FIGURE 3.4, and those for servers in FIGURE 3.5. Transaction calls use a data structure containing four items: a pointer to an 18-byte parameter area, a pointer to a signature (this pointer may be *nil*), the length of the main data buffer, and, if the length is not zero, its address. The layout of the parameter area is up to the Transaction Layer users. In requests, the parameter area will usually contain the private part of a capability which, combined with the destination port, forms a complete capability for an object.

Transaction carries out a transaction. The request, indicated by *preq* is sent to the service whose port is given in the *dport* parameter. The buffer available for the reply is indicated by *prep*.

```
struct tlparam {
        char      *tl_par;
        Port      *tl_sig;
        unsigned  tl_len;
        char      *tl_buf;
};

int transaction(dport, preq, prep);
        Port              *dport;
        struct tlparam  *prep, *preq;

putsig(psig);
        struct tlparam  *psig;
```

FIGURE 3.4. The Transaction Layer interface for clients

```
int getreq(port, preq);
        Port              *port;
        struct tlparam  *preq;

putrep(prep);
        struct tlparam  *prep;

getsig(gsig, func);
        struct tlparam  *gsig;
        void  *func();
```

FIGURE 3.5. The Transaction Layer calls for services

Transactions are blocking; that is, the calling process is blocked while the transaction is being carried out. A discussion on blocking vs. non-blocking communication calls can be found in § 4.1.3. Although transactions are blocking, an *interrupt*, caused by an external event, can temporarily unblock a process in a transaction. After examining the cause of the interrupt, the process may wish to abort the transaction by a call to *putsig*, which sends a *signal* to the service carrying out the request.

Services may receive these signals, or ignore them. In the latter case, sending a signal has no effect. In the former case, the service may decide to treat the signal as it sees fit. In both cases, the service must still return a reply and the client must expect one. The *putsig* call has one parameter: a pointer to a *tlparam* structure, of which only the first two fields are used for an 18-byte interrupt message, and, possibly, a signature.

The server calls are complementary to the client calls. *Getreq* initiates a transaction at the server's end. Its parameters are the port on which requests are to be received and a *tlparam* structure giving the address of the buffer for the received parameter area, the address for a received signature, and the length and address of the available buffer. *Getreq* blocks until a request is received.

Putrep has one parameter, a pointer to a *tlparam* structure, describing the reply.

Getsig can be used to indicate willingness to receive *signals* from clients. When a *signal* is received—indicated by an interrupt—the server can act on it by aborting the transaction and returning a reply immediately. Note that a *signal* is treated by a service as it sees fit, and that a reply is always required, *signal* or no *signal*.

3.2.4. The Transaction Protocol

The Transaction Protocol was designed to use a minimum of messages for sequences of short transactions. In longer transactions *control messages* are used to acknowledge receipt of requests or replies and to check for server crashes. In this section the Transaction Protocol will be described using *state diagrams*, but first the different message types and their layout will be shown.

The Transaction Layer uses three kinds of messages: *data messages*, *signal messages* and *control messages*. There are two types of *data messages*: *request messages* and *reply messages*, and there are three types of *control messages*: *ack messages*, *nak messages*, and *enq messages*.

The layout of all messages is the same, although the lengths differ; control messages contain only some of the fields. This is shown in FIGURE 3.6. *Control messages* and *signal messages* are 40 bytes long, and *data messages* between 40 and 32K + 40 bytes. The fields are used as follows: The 16-bit *length* field gives the length of the message (exclusive of the header); the *destination port* is the port where the message is sent; the *reply port* is the port where the message originates; the *signature port* contains the signature passed by the users (in *control messages* this field is unused); the *type* field contains the message type (0=request, 1=reply, 2=signal, 4=enq, 5=ack, 6=nak); the *sequence* field holds a sequence number, used in sequences of transactions; finally there is an 18-byte *parameter area* and the header is followed by a *data buffer* of length between 0 and 32K bytes.

We shall now show the protocol using the state transition diagrams for client and server end of the Transaction Protocol. The state transition diagrams for client and server are shown in FIGURE 3.7. The transitions have been labelled for easy reference in the text. In both diagrams two states are special: *START* and *FINISH*. These states indicate that no transaction record exists. An event in state *START* will cause a transaction to be started and a *transaction record* to be created. An event causing a transition to state *FINISH* will cause a transaction to end and the *transaction record* to be deleted.

The client states have the following meaning:

WTACK A request has been sent, the retransmission timer is running, and the client is waiting for an *ack message*, signaling receipt of the request, or a reply.

message length (2 bytes)
destination port (6 bytes)
reply port (6 bytes)
signature port (6 bytes)
type
sequence
parameter area (18 bytes)
data buffer (0 — 32K bytes)

FIGURE 3.6. Transaction Layer message layout.

WTREP The server has acknowledged receipt of the request. A crash timer has been started to detect server crashes.

WTENQ The crash timer ran off, or the client process sent a *signal*. A retransmission timer is running and the client is waiting for an acknowledgement on the *enq message* or *signal message*.

PIGTIM The reply has been received and a piggyback timer is running, delaying the *ack message* that must be sent to acknowledge receipt of the reply in case a new request is sent to the same server, which may also function as acknowledgement.

The states at the server end have the following meaning:

WTREQ The server process has called *getreq* and is now waiting for a request to come in.

PIGTIM An acknowledgement for a received request, *enq message* or *signal message* must be sent. A piggyback timer is running to delay transmission of the *ack message* in case the reply is sent quickly.

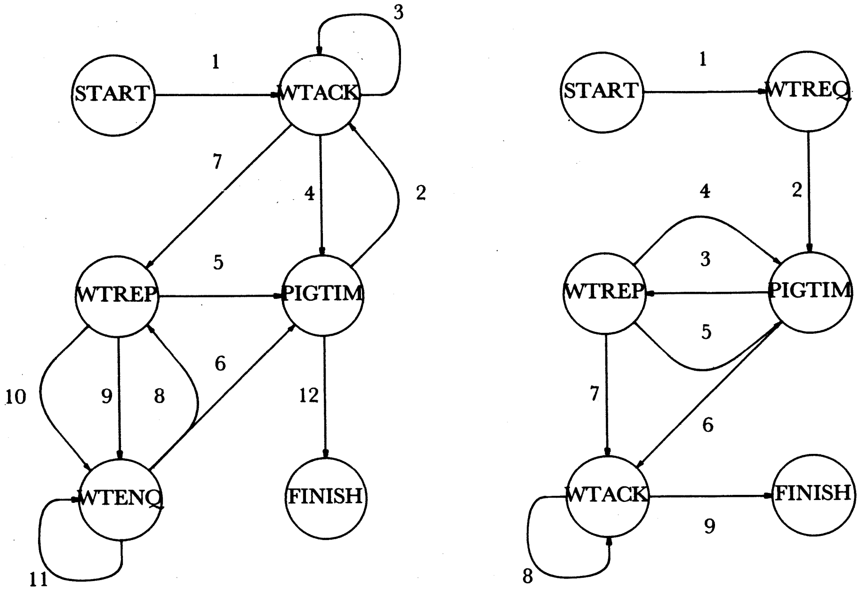


FIGURE 3.7. State transition diagrams for the Transaction Protocol, those for clients on the left, those for servers on the right. The transitions are discussed in the text.

WTREP The *ack message* has been sent. The Transaction Layer is now waiting until the reply will be sent.

WTACK The reply has been sent. The server is waiting for acknowledgement in the form of an *ack message* or another request from the same client. A retransmission timer is running in order to retransmit the reply, when necessary.

Now the meaning of the states is somewhat clear, the state transitions will be shown. The transitions are indicated by numbers in FIGURE 3.7, where one also sees the previous state and the new state. The *event*, causing the transition, and the *actions* to be taken when the event occurs will be given, first for the client end, then for the server end of the Transaction Protocol.

1,2 Event: The client makes a call to *transaction.*
Action: A check is made if there is an existing transaction record for the same server in state *PIGTIM*. If this is the case, this transaction record will be used after incrementing the transaction sequence number. Otherwise, a new transaction record is created, the sequence number is set to zero, and a unique port is generated to be used as reply port. Then a request

message is sent; if it is a new request it is sent using *putany*, otherwise it also serves as an acknowledgement and is sent using *put*. Finally, a retransmission timer is started and the crash timer is set to its initial value (but not yet started).

3 Event: The retransmission timer expires.
Action: The retransmission counter is incremented. If it has reached a predetermined maximum, the transaction is aborted, the client receives an interrupt, signaling transaction failure, and the transaction record is deleted. Otherwise, the request is transmitted again and the retransmission timer is restarted. For retransmission *put* is used. If this call results in a locate failure, another retransmission is carried out, this time using *putany*. All subsequent messages are sent using *put* or received using *get*.

4,5,6 Event: A reply is received.
Action: Timers are stopped, the reply is copied to the user buffers insofar as is necessary, an interrupt, signaling reception of the reply, is given to the client process, and a piggyback timer is started for the acknowledgement.

7,8 Event: An *ack message* is received.
Action: The retransmission timer is stopped, the crash timer is started.

9 Event: The crash timer expires.
Action: An *enq message* is sent, and the retransmission timer is started. The retransmission count is set to zero. The crash timer value is doubled, but not beyond a predetermined maximum value. The crash timer is not yet started, however.

10 Event: The client makes a call to *putsig*.
Action: The crash timer is stopped and set to its initial value. A *signal message* is made and transmitted. The retransmission timer is started.

11 Event: The retransmission timer expires
Action: The retransmission count is incremented. If it reaches its predetermined maximum, the transaction is aborted, the transaction record deleted, and the client process receives an interrupt signaling 'server crash.' If not, the *enq message* or *signal message* is retransmitted and the retransmission timer is restarted.

12 Event: The piggyback timer expires
Action: An *ack message* is transmitted, the transaction record is deleted.

For the server end of the Transaction Protocol the events and actions are:

1 Event: The server process makes a call to *getreq*.
Action: A *transaction record* is created, and a *getany* is done for the request.

2 Event: A request arrives.
Action: First a check is made if the request also serves as an acknowledgement for a reply in another transaction. If so, it is handled

accordingly (see transition 8). The request is copied to the user buffers (insofar as necessary) and the piggyback timer is started for the acknowledgement. A *get* is done for signals and retransmissions.

3 Event: The piggyback timer expires.
Action: An *ack message* is sent to the client (using *put*).

4 Event: A *signal message* arrives.
Action: If a *getsig* call has been made by the server, the parameters are put in the user buffers and an interrupt is caused. A piggyback timer is started in any case.

5 Event: An *enq message* arrives.
Action: start the piggyback timer.

6,7 Event: The server calls *putrep*.
Action: Timers are stopped and a *reply message* is sent to the client. The retransmission timer is started.

8 Event: The retransmission timer expires.
Action: The retransmission count is incremented. If it reaches the maximum number of retransmissions allowed, an interrupt is given to the server, signaling completion of the transaction and the transaction record is deleted. Otherwise, the reply is retransmitted, and the retransmission timer is once more started.

9 Event: An *ack message* is received or a new request arrives from the same client (see transition 2).
Action: An interrupt is caused signaling transaction complete, the transaction record is deleted.

Note the use of *get*, *put*, *getany* and *putany*. A new request is transmitted using *putany*, and received using *getany*, because, in principle, every service listening to the server port is capable of handling the request. Subsequent packets in a transaction are sent and received using *put* and *get*, because these have to go to the server process carrying out the transaction, not to another server process listening to the same server port. Some extra care has to be taken sending retransmissions: There are two possibilities, the request was lost or the reply (or acknowledgement) was lost. In the former case, the retransmission can be sent to any server process, but in the latter case the retransmission must be sent to the server process that carried out the request in the first place to guarantee that the request is not carried out more than once. The protocol handles this problem gracefully, by retransmitting using *put*. If the matching *get* cannot be found, the Transaction Layer is notified and the request can be retransmitted using *putany*. As an aside, note that messages are seldom lost, so this mechanism is not invoked often.

When the piggyback timer is set to zero, a special case arises. An acknowledgement is always sent immediately, without waiting for a quick reply. Although this costs a little network bandwidth, in some networks it results in a speed-up of the protocol as a whole. The overhead incurred by the piggyback timers is often greater than the gain in network delay. If the network is lightly loaded and fast, the immediate acknowledgement scheme is preferable over the piggyback timer scheme. The former is both faster and simpler.

Another way of speeding up the protocol is to use a special way of implementing timers. Before this is explained, note that timers (except the piggyback timers) are exclusively used to detect failures, and failures seldom occur in modern local-area networks. It is therefore not vital for the efficiency of the protocol that the timers work very accurately, as long as they work reliably. A '*sweep algorithm*' could be used to manage retransmission and detect failures. The algorithm has the following principle: The Transaction Layer causes the *sweep algorithm* to be executed periodically. Whenever the algorithm is executed, it checks whether any transactions have become *stuck*, by comparing their state with their previous state. If a particular state has not changed in the period indicated by its timer, the appropriate recovery is started (*e.g.*, retransmission, failure report, sending an *enq* message). The Transaction Layer thus incurs no overhead starting and stopping timers whenever an event occurs. The overhead is in the *sweep algorithm*. The overhead of handling events increases with increasing network traffic, while the overhead of the sweep algorithm is nearly constant.

3.2.5. Conclusion

Although the transport layer of most networks provides connection-oriented service to higher layers, we believe that in local networks there is much to be said for a transaction-oriented service instead. A proposal for such a service has been described in the context of a capability-based distributed operating system, but the general idea holds equally well for a system that does not use capabilities. The basic idea is to provide the user of the transport service with semantic primitives for carrying out reliable transactions, each one (in principle) independent of all other ones. Using the protocol described, reliable communication can be achieved under conditions of heavy load with only two messages per transaction, one in each direction. When transactions are widely separated in time, a third message is needed so the client can tell the server to release its transaction record.

3.3. The Directory Server

In a capability system, users must store a capability for each object they access. The number of capabilities held by any one user can become quite large; hundreds, or thousands of capabilities will not be uncommon. Management and storage of capabilities is not a trivial problem. The Directory Server is an attempt to solve this problem.

In brief, the Directory Server provides to its users a mapping of easy-to-remember string names onto capabilities. Such a map is called a *directory*, and there can be many of them. A capability is needed to search or change a directory. This capability may be stored in another directory. The collection of directories thus forms a directed graph, which may be searched and changed by the Directory Server's clients.

Capabilities are precious objects, and it is important that they do not fall in the wrong hands. The Directory Server must therefore be made particularly secure against all possible forms of attack. For one thing, it can not leave information on files in plaintext, because a file system crash, or a concerted attempt to break into the file system might reveal sensitive capabilities.

The huge number of capabilities that users can collect presents the designer of the Directory Server with the problem of finding convenient mechanisms for Directory Server clients to store capabilities, pass them to other (selected) users, and retrieve them easily and efficiently.

3.3.1. Directories

The Directory Server's main task is to map **names** of objects into **capabilities** for them. Names are collected in **directories**. A client, wishing to find the capability associated with some name, must present that name and indicate a directory (or possibly a set of directories) to be searched.

In principle, a Directory Server could maintain one huge list of names to be searched, but this is inconvenient for two reasons. First, all names have to be different, so no two users may refer to their objects by the same name. Users are therefore not free to choose any name they like for an object, which may cause confusion. Second, if one list is maintained, measures need to be taken to prevent every user from searching the whole list where other user's private capabilities can be found.

Each user can maintain a set of directories to store (*name*, *capability*) pairs, using any convenient name. Different directories can be created for different types of objects, or different directories can be created for references to objects related to various projects. By giving an appropriate capability for a directory to another user, it is possible to share objects easily. Entering a capability for an object in such a shared directory allows all users with access to that directory to manipulate that object according to the rights in its capability.

Like other objects, directories are also accessed using capabilities. No operation on a directory is allowed, unless an appropriate capability can be presented. In fact, it is not possible to refer to a directory other than through its capability. Different operations on a directory need different rights in the presented capability. In the next section the operations on directories are discussed.

Since directories are referenced by capability, it is possible to enter a (*name*, *capability*) pair for one directory in another. Not only can a hierarchy of directories be thus constructed, it allows directories to be linked together to form a general directed graph; we expect to demonstrate the usefulness of this concept in a later section in this chapter. The directed graph structure allows a user to search one directory for the capability of a second, in order to find a capability in a directory several *nodes* and *arcs* away. As a shorthand for such operations, the Directory Server allows **pathnames**, sequences of names, that all but the last name refer to directories.

Although only one design of a Directory Server is discussed here, several different types of Directory Servers could coexist in an open distributed system such as ***Amoeba***. Clients of the Directory Server will be aided if all Directory Servers cooperate. It should be possible that, for instance, pathnames can transparently snake back and forth across directories in several Directory Servers. In order to achieve this, Directory Servers must know the *search directory* syntax used by the other Directory Servers so they can forward partially expanded pathnames. The simplest way to do this is to use the same syntax in all Directory Servers for directory searches.

3.3.2. The Directory Graph

Directory capabilities can be stored in other directories, forming a directed graph of directories. Every user, when given access to the system for the very first time receives the owner capability for a **home directory**. Every time the user logs in, the home directory capability is given to the user. (For security, it could be stored in encrypted form, with the user as the only holder of the decryption key.) In his home directory, each user keeps capabilities for several public directories. Here capabilities can be found for objects that can be accessed by anyone in the system. The user can also maintain special directories where capabilities for objects can be stored that can be shared with other users. A search capability for that directory can then be written in a public directory. An example of a fragment of the graph structure that can thus be formed is given in FIGURE 3.8.

The general graph structure that can be formed with directories provides their users with a general mechanism for storing capabilities which allows sharing of objects, keeping objects private, and communicating capabilities. Many types of sharing can be implemented trivially. Two users can share a directory by both entering the owner capability for it in a directory of their own. Another form of sharing, which is shown in FIGURE 3.8, is user *cogito*'s *share* directory. In this directory, *cogito* can make entries for capabilities that other users may use, such as the *read-only* capabilities for file *file-a*. Not shown, but also easy to implement is a mail service: A search capability is made public for a mail directory that contains *append-only* capabilities for files for each user wishing to receive mail, and each user receives an *owner* capability for the file with his mail.

Although the Directory Server, as presented, is versatile enough, it has the drawback that sharing objects is rather inefficient. In an open user community, where most users allow read-access by other users to their objects (mostly files

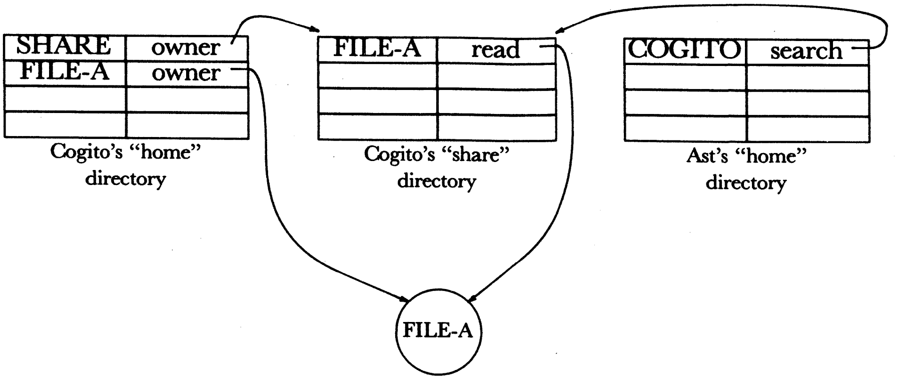


FIGURE 3.8. Part of a directory graph. User *cogito* uses a separate directory to facilitate sharing of objects, such as the file *file-a*.

and programs), the operations necessary to enter different capabilities for each object in a number of different directories, each time an object is created, may be too much of a burden on the user. An alternative mechanism for easy sharing of objects which is equally versatile would be desirable here.

Such a mechanism can be constructed, using an alternative directory structure. A directory structure is used that does not have just two columns for names and capabilities, but three or more. The first column contains names, the others capabilities. In the original approach the Directory Server issues one search directory capability, in this novel approach the Directory Server issues a different search capability for each column. Clients make entries, not just of (*name, capability*) pairs, but of (*name, capability, capability*) triples, or even (*name, capability, capability, capability*) quadruples, depending on the number of columns.

The mechanism is used as follows: The owner of a directory (one with two *capability* columns in this example) will use the first *capability* column for *owner* capabilities for his objects, the second *capability* column will be used for capabilities to be shared (*e.g.*, read capabilities for files, (column two) search capabilities for other directories). *Owner* capabilities for subdirectories will be put in column one, search-only capabilities will be put in column two. In this way, two or more directory structures can be superimposed. The owner has full access to every object via the capabilities in column one. Others can get restricted access via the directory graph for column two. To this end, they receive a special capability with a bit in the *rights* field that gives access to column two.

When a directory is created, the number of capability columns in it must be specified. The directory owner has full control over the contents of all columns. A special *null* entry can be made in one of the columns, making the whole entry (including the name) invisible to a holder of a search capability for that particular column. An example of such a directory structure is shown in FIGURE 3.9.

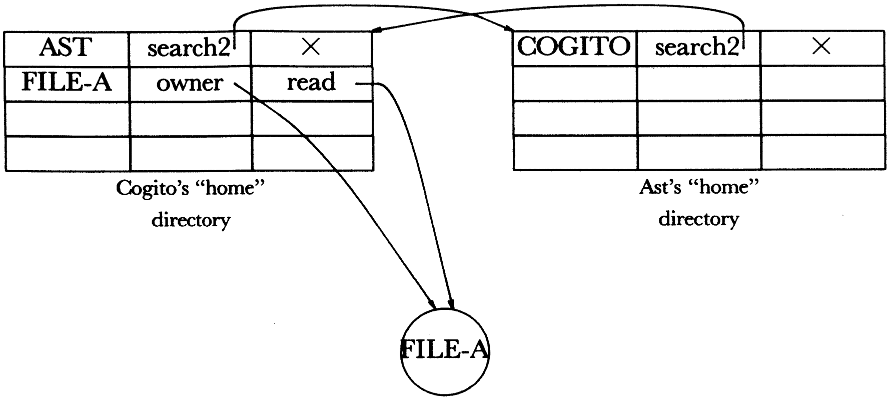


FIGURE 3.9. A three column directory allows easier sharing of objects.

3.3.3. Directory Server Operations

Apart from directory search, the Directory Server must implement many more operations on directories and directory entries. Typical operations are: list the contents of a directory, create a directory, remove a directory, restrict rights to a directory, enter a (*name*, *capability*) pair in a directory, remove such a pair, change the name on an entry, change the capability in an entry, etc.

Several rights are necessary on Directory Server capabilities so clients can be allowed, for instance, to search a directory without being allowed to change it.

3.3.4. Protection

It has already been mentioned that the security of the Directory Server is particularly worth while breaking, due to its precious information contents. A Directory Server's armour must be especially strong to withstand not only accidental leaks of confidential information, but also concerted attacks against its security. Leaks lurk in various places, so the Directory Server must be designed extra carefully.

There are two ways for malicious users to illicitly obtain capabilities from other users. One way is to steal them directly from the users, by finding out passwords, breaking into programs, luring them into giving them away, etc. The other way is to break the Directory Server's security, for instance, by reading the disk where the Directory Server stores capabilities, by sending cleverly doctored sequences of requests that yield capabilities that ought not to be handed out, by breaking into the Directory Server's address space (*e.g.*, by forcing a core dump), etc. In this section we are mainly concerned with the security of the Directory Server itself.

The security of the Directory Server can be attacked at several points. One form of attack that can be tried by anyone is to try to formulate requests, or sequences of requests that yield a capability to which the attacker has no right. It is not very difficult to design a Directory Server interface that allows no 'funny business,' but bugs in the Directory Server software may cause the Directory Server to deviate from its specs. Actually, this form of attack has frequently been used successfully to break the security of operating systems.

The second form of attack is assailing the file system where the Directory Server puts its information. Under normal circumstances it can be assumed that file servers are secure, but surrounding most (secure) file systems there is often a great deal of insecurity. This insecurity can be caused by simple disk errors, which affect the integrity of the file system. Due to such errors, it is possible for a user to suddenly find parts of other files embedded in his own. Hardware failures may cause information to be revealed accidently. Backups have to be made of the file system; these backup tapes are often stored where everyone has access to them.

The information stored by the Directory Server on disk is too sensitive to allow an inquisitive user to obtain that information just by reading the disk. An obvious solution to this problem is to have the Directory Server encrypt the information it puts on the disk. An intruder, illicitly obtaining the contents of a file, or a disk drive with many files, is not capable then of retrieving capabilities without the key. Naturally, the Directory Server must carefully guard the encryption key, or keys, and definitely not store them on a file.

Encrypting the Directory Server data on the disk already makes it much more secure, but there is still a third form of attack that cannot be withstood in this way. The Directory Server itself can also be subject of an attack. If an intruder has access to the code and data of a Directory Server process, then that intruder has access to the Directory Server's secret keys, and the capabilities that the server uses to access the files where its sensitive information is kept. It is not wise to assume that an intruder has no access to the Directory Server process' code. To depend on keeping programs secret for security is to court disaster. Programs have listings; listings lie around in offices in reach of almost anybody. Even if the program code could be kept secret, there is no reason why the designers and programmers of the Directory Server should be in the privileged position of being able to get at what it stores. In cryptography, the strongest cryptographic systems are considered to be those that do not rely on the cryptographic algorithms being secret. In the case of secure services, this principle also applies: stronger security is achieved if the security of the service does not suffer when the programs are made public. Tampering with the Directory Server's binary can be preventing by checksumming the binary and storing the (encrypted) checksum elsewhere. When the Directory Server is started, the checksum is fetched and the code checksummed.

Another mechanism is needed to make the Directory Server secure. The keys that the Directory Server uses to generate its server port and encrypt the data it writes on file must not be present in the program, but when the Directory Server is started these keys should be given to it. They can, for instance, easily be derived from a secret password that the Directory Server's manager types in, or better still, derived from two secret passwords that two managers type in.[64] In this way, even the Directory Server's managers can not break the security of the Directory server, unless they cooperate. The Directory Server would then work as illustrated in FIGURE 3.10.

```
Dserver() {
        /* Get passwords from the managers: */
        GetPasswords();
        /* Use the passwords to generate the server get-port */
        GeneratePort();
        /* Check the port by computing the put-port and
         * comparing that to the public Directory Server
         * put-port: */
        CheckPort();
        /* Generate the key that is used to encrypt the data
         * on disk from the passwords:
         */
        GenerateKey();
        /* Now the server is ready to process user requests: */
        for (;;) {
                getreq(...);
                /* process the request */
                ...
                putrep(...);
        }
}
```

FIGURE 3.10. An example of a possible initialisation for the Directory Server.

In spite of the mechanisms described above, one form of breaking the Directory Server's security remains. If the server's memory contents can be examined, the keys the server uses are revealed. In a properly working operating system, unauthorised processes have no access to other processes' memory contents, but a server crash, perhaps causing a core dump, may be exploited to get at the secret keys. But, happily, there are ways to overcome even this problem.

The solution proposed here uses one-way functions. When a directory is created, the Directory Server creates a key that it uses to encrypt the capabilities stored in it. This key is embedded in the capability for the directory and given to the client. The Directory Server does not store the key, but stores the result of applying a one-way function to it. In this way, the Directory Server has the means of checking capabilities for directories for validity, namely by applying the one-way function to the capability and comparing to its stored capability, but it does not have the means to extract stored capabilities from a directory without the client capability for that directory. Since the Directory Server is not capable of reading its own directories without the client capability for them, an intruder will certainly not be able to.

4

PROCESS MANAGEMENT

Traditionally, process management was solely carried out by the operating system. Process creation, process scheduling, exception handling, it was all done by the operating system kernel. Slowly this tradition is changing, and for good reasons: the central position of the operating system kernel in a computer system is crumbling, the 'open system' philosophy is taking over. More and more functions, traditionally provided by the operating system kernel itself, are now provided by 'services,' usually implemented as ordinary user processes. This has not only led to the realisation of smaller, simpler, and easier-to-maintain operating system kernels, but has also given programmers more flexibility in choosing the type and complexity of the services they want themselves; the choice is no longer made by the system designers.

However, the operating system kernel cannot be completely dispensed with. Hardware interrupts and software exceptions must be handled, on multiprogrammed machines, processes must be protected from one another, and an interface must be provided for processes to communicate with each other and with other parts of the system. Process management must—at least in part—be carried out by the operating system kernel, but for flexibility, functions traditionally inside the operating system should be provided through services, implemented in user space, where they are easier to maintain, and, when not needed, will not hinder the user, just by being there.

Making out which functions must be provided by the kernel, and which functions can be implemented in user space is complicated. Conceptually it is nice to have as many functions outside the kernel as possible, but, if this makes the system unacceptably slow, no one is likely to use it. Tradeoffs will likely be necessary between fast kernel functions and flexible, but less efficient user services.

Before such considerations can be made, an inventory must be made of the process management functions needed, and the execution environment must be designed that allows an efficient, yet simple, style of programming. This is easier said than done. During the design phase of the ***Amoeba*** distributed operating system, the execution environment has been changed more often than any other part. The success of an operating system kernel depends critically on the execution environment it provides, and how efficiently it is provided.

These considerations determine the presentation of the remainder of this chapter: the first section discusses the design of the process execution environment, the second section deals with the support that the kernel provides for this execution environment, and the last section describes supporting services, such as *process service*, *loader service*, and *bootstrap service*.

4.1. The Execution Environment

Programmers have become used to the kind of execution environment provided by centralised operating systems. For the most part, this environment is determined by the host computer: its instruction set, the size of its address space, the presence of segmentation, etc. This aspect of the execution environment is inflexible, fixed by the computer manufacturer. Fortunately, only few programmers still interface to a machine at this level. Most programs are written in 'high level languages'* that hide this aspect of the execution environment from the programmer. The instruction set is therefore not an important factor of the execution environment.

An aspect of the execution environment that affects the programmer much more is the interface provided for communication with the outside world, and the primitives for process management. In the next sections we shall focus on the operating system interface, and interprocess communication.

4.1.1. Open Operating Systems

The term *'open operating system'* was first used by Lampson and Sproull,[36] to describe an operating system that can be thought of as (we quote): "offering a variety of facilities, any of which the user may reject, accept, modify or extend." This can be contrasted with the philosophy underlying traditional 'closed' operating systems, which provide one set of facilities that the user must accept, and cannot modify. Most operating systems fall in the latter category, and it is a useful exercise to compare the pros and cons of traditional 'closed' systems to the currently emerging 'open' systems.

A closed system provides one set of services, thus enforcing a degree of standardisation. The interface to these services is the same for all users. Mechanisms for authorisation, object naming, service access may be provided in a

* Some claim that only *Algol68* merits the title 'high-level language,' while others maintain that even BASIC deserves to be called a 'high-level language.' Both are extremist views.

uniform manner. The hardware of many computer systems, which is often difficult to use, is hidden by the operating system to make the machine easier to use. Protection and security are easier to provide, because there is one well-defined user interface. Closed systems are often implemented by putting the facilities it offers in the operating system kernel, resulting in a large kernel which is difficult to maintain and will never be quite bug-free.

Open systems, in contrast, (potentially) provide a wide range of services for each operating system function. Users may reject the services offered by the operating system in favour of their own. It is not required that services are accessed in a uniform manner, although a particular access mechanism may become standard as a matter of general convenience. Traditional operating systems often make it impossible to use special features provided by the hardware, or access peculiar peripherals; an open system can provide mechanisms to hide the painful behaviour of the hardware from one user, while allowing another access to its interesting features. In an open operating system, only the essential services are in the operating system kernel. The kernel can thus be kept small, its maintenance becomes easier, and the services themselves, each being implemented as a separate user program, are easier to maintain also. Furthermore, a service crash affects the users of the service, but does not crash the system as a whole.

In many applications, the one-service-for-everyone approach of closed systems has caused considerable inconvenience. Particularly illustrative of this problem are the attempts to put database management systems on top of existing operating systems.[70, 73] In most cases this results in inefficient database management systems, and the main cause for this inefficiency is that the operating system provides *more* service than is needed by the application. The operating system has a standard way of doing things and this does not allow the database management system a fine enough grain of control. A few of these unneeded services are:

(1) In-core buffer caches of recently used disk blocks do not work because database systems usually have insufficient locality of reference to make them work. Database management systems usually know which blocks they will likely use again, so they maintain their own disk block caches.

(2) Many operating systems do read-ahead, thus trying to have the next section of the file ready when it will be read. Database management systems seldom read files consecutively.

(3) Crash recovery is difficult to implement if the underlying operating system does not immediately put blocks on disk when they are written by the database management system.

(4) If a database management system is forced to use the flat file system provided by the operating system, it has no control over the placement of blocks on disk resulting in many unnecessary seeks over the disk. Furthermore it introduces an extra level in mapping keys onto disk blocks.

An open operating system can provide a set of facilities that database management systems can choose from. A database management system does not have to be burdened by a collection of unnecessary and counterproductive facilities, it may even find some useful facilities that traditional closed system cannot provide.[73] An example of such a service is the Amoeba File Server.[46]

The point that is being made here is not that file services should not provide read-ahead or maintain buffer caches for efficiency; the point is that there are some applications that are severely handicapped by such 'features,' so it should be possible that several types of file service can co-exist, one for each type of application. This is not possible if file service is an integral part of the operating system: A file service that caters for every kind of application would be too big, too inefficient for simple applications, and too difficult to design and maintain. Furthermore, if a new type of application came along, it would not be possible to adapt the file service. Obviously, as many services should be removed from the operating system kernel as possible, and provided as stand-alone user programs. File service has served as an illustration only; the arguments apply to most types of services.

4.1.2. The Kernel Facilities

An open operating system must provide the tools that allow the users of the system to implement operating system services. These services, such as file service or terminal service, must be in the user domain whenever possible, the system only providing the means that allow communication with services, and the mechanisms that prevent processes from interfering with each other.

The kernel is that part of the operating system that is outside the reach of the system's users. True to the open system's philosophy, it must provide a minimum of functions, allowing most operating system facilities to be built in the form of user processes. Its first task is process management. It must provide the mechanisms for process creation, execution and termination. Furthermore, the kernel must protect processes from each other. One process should not be allowed uncontrolled access to another process' address space or disrupt its normal functioning. The second task of the kernel is to provide access to peripherals at a convenient interfacing level. In some cases this may imply providing access to the hardware directly, in others, providing access through a device driver. When the kernel is multiprogrammed, peripheral access must be controlled by a protection mechanism; not every user process should be allowed access to, for instance, the disk. The third task for the kernel is to allow processes to access the services provided outside the kernel; that is, provide an interprocess communications facility.

Portability of kernel software is an important property. While using the facilities offered by the hardware optimally, the kernel must be designed so it can be ported to other hardware reasonably easily. This requires writing kernel software in a high level language and separating the machine dependent parts from the machine independent parts. Although the portability principle is widely accepted and few operating systems are written in assembly language any more, the principle is important enough to mention here anyway.

Although the kernel is outside the user's reach, it is still important that even the kernel be structured in a modular way. It is, for example, desirable that on some of the larger CPUs there should be the possibility of using multiprogramming to put the available computing power to better use, while on the smaller microcomputers multiprogramming is often not possible due to the absence of suitable hardware. A highly modular operating system kernel could be equipped with 'plug-in' multiprocessing modules, 'add-on' device drivers, etc. From the viewpoint of portability of software this is an important principle: one set of device drivers suffices for both monoprogrammed and multiprogrammed machines, and it allows, for instance, memory management using paging on one host, and segmentation on another without a need for changing the rest of the kernel.

While many operating system kernels exist today, there are still relatively few kernels especially designed for use in an open operating system environment. Let us take a look at some of the operating system kernels that function in a more-or-less open environment.

It is our contention that an operating system kernel for an open system should provide an absolute minimum of facilities to its users: an environment for process execution which allows efficient communication between processes and between processes and peripheral devices. Naturally, this may not be done at excessive cost: If it is not possible, for instance, to build efficient and secure protection mechanisms outside the kernel of the operating system, but trivial to do so inside of it, those mechanisms should by all means be put in the kernel. We hope to show, however, that with a minimum of kernel facilities it is still possible to build efficient systems.

As an aside, it is worth noting that the less reliance there has to be on an operating system kernel in a distributed system, the more freedom can be given to the participating systems. We explicitly allow both private personal computers and existing operating systems to take part in the ***Amoeba*** distributed system, provided those systems adhere to the protection mechanisms discussed in Chapter 2.

4.1.3. System Calls for Interprocess Communication

There are as many different interprocess communication mechanisms as there are operating systems; no wonder, since there are many ways to set up an interprocess communication system: it can be made *stream* oriented or *datagram* oriented, or something in between. The calls can be made *blocking* or *non-blocking*, communication can be *unidirectional* or *bidirectional*. These options can be arbitrarily combined, and all of these options have subtler suboptions.

Usually, when there is a choice of many options, most can be discarded immediately, either because they are impractical, or because they are infeasible, or because they are inelegant. Unfortunately, however, most of the combinations mentioned above are in use, with sensible arguments supporting their choice. The weight attached to different arguments determines the choice of mechanisms.

We have chosen a capability-based protection using ports, because it allows an implementation of a protection mechanism without having to rely on a secure operating system kernel, and, because it allows addressing services without needing a separate *name server* to map server names into *process-id*s or machine numbers. (See Chapter 2, in particular §§ 2.3 and 2.4.)

We have chosen a message-oriented transaction mechanism because it is a better semantic model for distributed operating systems than a connection-oriented mechanism and because it can be implemented much more efficiently than virtual circuits, especially when measurements show that most network traffic consists of short exchanges of information increasing the relative overhead of opening and closing connections to an unacceptable level. (See § 3.2.)

We have chosen *transaction* type communication, because most interactions between client and server processes have a transaction nature: A client wants the server to do something and sends the server a request; the server carries out the request and returns a reply to the client. Furthermore, transactions can be implemented efficiently and provide a natural mechanism for synchronisation of processes. (See § 3.2.1.)

We have chosen *blocking* calls; that is, a client that does a transaction (send a request and wait for a reply) is blocked until the transaction completes; a server that does a *getrequest* (wait for a request) is blocked until a request is received.

This last choice is not at all obvious. From a client's point of view, blocking transactions resemble a *remote procedure call* mechanism.[68] Nonblocking transactions allow a client to start a transaction and continue doing useful work while the transaction is being processed by the server. When the transaction finishes, the client is warned (*e.g.*, by means of an interrupt).

Nonblocking system calls allow the programmer to exploit the possibilities of parallel processing: Requests to remote services can be processed while a client process continues doing work. A client process can, in fact, have multiple outstanding requests. Likewise, non-blocking interprocess communication gives a server process the freedom to handle many requests simultaneously: A server can start by doing multiple *getrequests*, and, when an interrupt occurs, start processing the received request. During this process it may become necessary to call on a subservice, and while the server waits for the reply from a subservice, it can continue handling other requests.

Blocking system calls allow no such freedom: A client process is blocked while waiting for the reply from a server, so it cannot do any work while the server is busy, nor can it send requests to more than one service simultaneously. Server processes can only handle one request at a time, also because they block on subtransactions, and because they can do only one *getrequest* before being blocked.

In the first implementation of the ***Amoeba*** Transaction Layer, we decided to implement non-blocking calls. A call made by a client process to *transaction* * causes the run-time system to send a request and await a reply. The client process would continue execution, and an interrupt, caused by the run-time system would finally warn the client that the *trans* call had completed. A server process, analogously, would call on *getrequest* * which caused the run-time system to prepare for reception of a request; the server process would continue to execute and the run-time system would warn the server that a request had been received. After processing the request, a reply would be sent by a call to *putreply*.

Although this non-blocking method gave processes more control and allowed more parallelism, it had a severe drawback: Its exploitation requires very skilled programming and often leads to the introduction of race conditions, due to interrupts that occur at inopportune moments. An even greater problem is how to structure server programs to handle more than one request simultaneously in a manageable fashion. We found two approaches to structuring server programs.

```
HandleEvent(event, params) {
    /* This routine is called when an event occurs */
    TransRec = FindTrans(event, params);
        /* Find the transaction record for
         * the transaction that caused the event
         */
    (*EventTable[event][TransRec->state])(event, params, TransRec);
        /* Call the routine to do the work from a table
         * of pointers to functions, indexed by event and
         * transaction state
         */
}

MainLoop() {
    /* Main loop of the server */
    for (;;) {
        Pause();
        /* wait for an event */
        HandleEvent(e, p);
        /* Handle the event; the e and p parameters are set
         * by the interrupt routine
         */
    }
}
```

FIGURE 4.1. Server process structure in a finite-state implementation

One approach is to write a server process as a finite state machine: An event (*e.g.*, the reception of a request from a client or a reply from a subserver) causes

* See § 3.2.3

the server to find the *transaction record* for the transaction that caused the event, carry out an appropriate action, based on the contents of the transaction record and the information given by the event, and change over to a new state. Every time a new request arrives, a new transaction record is created. Whenever a reply is sent, a transaction record can be removed. This method of programming—although feasible—does not provide the programmer with a clear view of the structure of the server's algorithm: the algorithm is split up into pieces, separated by the points where an event must be awaited. With this approach it is, for instance, impossible to use subroutines to make a layered implementation of the server's algorithms; the structure must always be like the one depicted in FIGURE 4.1.

The other approach is the following: Using a small run-time library package, called **task manager**, a server process is split into a number of subprocesses, called **tasks**. The tasks share the address space, and each task has its own stack, stack-pointer and program counter. All system calls are trapped by the *task manager*, which passes them on to the kernel. System calls are non-blocking, so the task manager regains control immediately after making the call to run another task, if appropriate. To the task itself, system calls appear blocking. When a system call completes, the calling task resumes processing as if the call were blocking.

A server process can thus be organised into a number of parallel tasks; each task executes one client request at a time, and carries out system calls as if they were blocking. This makes efficient structured programming possible, in contrast to the previous method, which led to programs without much structure that were very difficult to understand and maintain.

The task manager contains the only critical code, since task switching only occurs when a task makes a system call. There is no danger of race conditions if tasks leave internal tables in a consistent state before making system calls. Getting the machine-language assist to work correctly is tricky, but it has to be done only once.

We have used this second approach in a simple disk server and file server, written by two students in less than 6 months. The task structure proved to be a valuable approach to managing multiple requests in one server process. Once the task manager worked correctly, all attention could be given to the design of the server itself, administration of concurrent requests required only little concern.

The success of the second method for managing concurrent requests made it clear that, from the viewpoint of program design, parallelism obtained through parallel processes (or tasks) is preferable to parallelism obtained through non-blocking system calls. This is a strong argument for the design of an operating system interface that allows management of parallel processes sharing an address space, combined with blocking system calls.

4.2. The Amoeba Kernel

Putting the theories of the previous section about distributed operating system kernel properties to the test resulted in the design and implementation of the Amoeba Kernel. Several versions have been tried before the kernel as described here was designed.

The primary design considerations have been to construct a kernel that is as small as possible, provides a simple, yet powerful, process interface through blocking interprocess communication system calls, and 'cheap' processes for parallel processing.

The concept of a *task* is used as a model for both the modules of the kernel, which operate in parallel, and sets of processes, sharing an address space, which also operate in parallel.

4.2.1. Clusters and Tasks

To avoid confusion with the familiar concept of a process, the entities that make up an ***Amoeba*** system will be called *clusters* and *tasks*. A **task** can be viewed as a deterministic process, communicating with other tasks via shared memory or *transactions*. A **cluster** is a group of tasks sharing one address space.

The whole of the address space is shared, except the stack. The operating system kernel controls these processes by executing their system calls and scheduling them.

Since several tasks can share one address space, there could be race conditions when two tasks simultaneously access a data structure. If pre-emptive scheduling of tasks occurs within a cluster, race conditions must be avoided, *e.g.*, by semaphores, monitors, or message exchanges. Another synchronisation method would be to use non-pre-emptive scheduling between tasks in the same cluster; that is, tasks within a cluster run until they are blocked before another task in that cluster is allowed to run. Fortunately, it is not difficult to use scheduling techniques, where there is pre-emption between tasks in different clusters, but not between tasks within a single cluster. As a matter of fact, the Amoeba Kernel uses this last scheduling technique.

In practise, most clusters consist of only one task. This reflects the conventional notion of a process. Services, however, will often be implemented as a collection of identical tasks. Typically, each task, after initialisation, will block on a *getreq*. When a request comes in, one of the tasks will become runnable, and execute the request. While processing requests, it is possible that other services must be invoked. Appropriate transactions are made to subservers, temporarily blocking the task. When the task has carried out the request, it will send a reply and go back to the top of the loop. Within a single task, there is thus a single thread of code, and its execution will be 'deterministic.' The cluster as a whole handles many requests simultaneously. Tasks will run until blocked, other tasks, after having become unblocked, taking their place.

Tasks of a cluster share an address space. This makes it easy to share global variables, caches of data, pools of buffers, etc. Tasks must however, leave these variables in a consistent state before blocking, but this is not difficult.

4.2.2. Kernel Tasks

Conceptually, a number of concurrent activities take place inside an operating system kernel—both for single-user machines or for multiprogrammed machines. Interrupts especially are a well-known source of headaches for programmers. A widely used technique for keeping this parallelism under control is *message passing*, as used by Brinch-Hansen[6] for instance. In this technique, the kernel is divided into a number of semi-independent processes which exchange messages for synchronisation. These processes run to completion before another process is allowed to run, thus avoiding race conditions. Hardware interrupts are immediately converted to messages and queued for the appropriate kernel process.

The Amoeba task concept lends itself naturally for structuring the kernel into a number of parallel tasks. Kernel tasks communicate by means of transactions with the same semantics that other tasks and processes use.

4.2.3. System Calls

Traditionally, operating system services are accessed using system calls, traps from a user program into the operating system kernel, parameterised to tell the kernel what to do. In open systems, many services will be provided by user programs, while others are still offered by the kernel itself.

Exactly what services must be provided by the kernel, and what services can or must be provided outside it, depends on many things: whether it is possible to implement the service inside or outside the kernel, whether an implementation outside the kernel will be unacceptably inefficient, or whether an implementation inside the kernel is too inflexible.

Evidently, the decision which services to implement inside the kernel, and which to implement outside it, must not be enforced by the operating system interface: the system calls for accessing service X must be the same whether the service is a kernel service, or a user service. Thus, services can be implemented in several ways, without having to make changes to the user interface.

All this implies that operating system services must be accessed like any other service: by making transactions. The only system calls available in the Amoeba kernel are therefore the calls provided by the Transaction Layer, *trans*, *getreq*, *putrep*, etc., as described in § 3.2.3. The implementation of these calls is sketched in the next section.

4.2.4. Transaction Layer Implementation

Transactions form the sole mechanism for interprocess communication; transactions are also the mechanism by which a process communicates with the operating system. Although this interface to the system is flexible and conceptually simple, it can only be practical if it can be made fast enough. ***Amoeba*** will not be used if a conventional operating system does the requested work an order of magnitude faster, even if its operating system interface is ugly. Obviously, if one thing affects the speed and efficiency of a message-passing operating system, it is the message-passing mechanism. The transaction mechanisms form the basis of the system, fortunately; this allows the kernel to be designed around optimally functioning interprocess communication mechanisms.

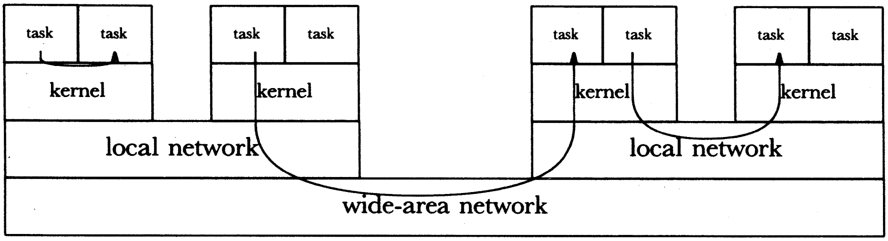


FIGURE 4.2. Local interprocess communication must be handled locally: communication between tasks of a cluster is handled by the task manager; communication between processes of one kernel is handled by the kernel, etc.

Heavily used services must be provided close to where they are used. It is therefore logical to optimise the mechanisms for local transactions. This implies that transactions local to one cluster, *i.e.*, between tasks, can best be handled by that cluster's task manager; transactions local to one operating system should be handled by the operating system; transactions local to a local area network must be handled by the network, and so on. This is illustrated in FIGURE 4.2.

There is a hierarchy of transaction handlers that works as follows: When a message (request or reply) is offered to it for transmission, the transaction handler checks if its destination is local to it; if so, it handles the message itself, otherwise it passes the message to its superior. To detect local message destinations, each transaction handler keeps a table of local ports and destinations. Assume thus, that a process carries out a *getreq* operation. The *get-port* is then entered in the local table, and the *getreq* is passed to the superior transaction handler. After this, a process carrying out a *trans* with a *put-port* that matches the *get-port* can be handled locally. The *reply-port* is entered in the table, so the reply can also be handled locally.

In principle, every *getreq* must be passed on all the way to the top of the hierarchy, because it is not known by the system where the processes that will send to it are. Many services do know this, however, so, as an optimisation mechanism, an extra call to the transaction layer can be implemented that allows a service to tell the system 'how local' its clients are. The operating system kernel, for instance, offers some services, purely to local processes. *Getreq* calls, made by the operating system for local services, need not be passed to the network. The *local* calls tells the Transaction Layer that a port is (1) local to the cluster, (2) local to the host, (3) local to the local-area network, or (4) global. By default, a port is always assumed global.

Mention must be made of the mechanisms for handling *interrupted transactions*. A server process can make a call to *getsig*, to tell the Transaction Layer that it is prepared to be interrupted by its client. This is particularly useful in services that execute requests that take a long time, as, for instance, a terminal server, an editor service, or an alarm-clock service. Interrupts are received as *signals*, header-only messages sent by a client to a server. Client processes can only send them while engaged in a transaction with a server; server processes can only receive them while engaged in executing a client request.

When a signal arrives, a server process can either be processing the request itself, or it can be waiting on the reply from a *subtransaction* with a subserver. In both cases the interrupt routine is called (the address of which was specified in the *getsig* call). From the interrupt routine, no transaction calls are allowed, except a call to *putsig* to interrupt an active subtransaction. There are two calls to return from the interrupt routine: *rti* and *longjmp*. *Rti* returns to where the server process was at the time the interrupt occurred, and, if the server was blocked on a subtransaction, continues blocking. *Longjmp* jumps to a position previously indicated in a call to *setjmp* (much like the calls of these names in the *C* run-time library), and, if necessary, breaks off a subtransaction by unblocking and ignoring the reply (if it arrives at all).

The mechanism for handling interrupts can thus be used to propagate an interrupt through a hierarchy of services. When a user hits the *interrupt* key of his terminal, for instance, the terminal server is interrupted. The terminal server passes the interrupt on to its client, for example, to the *shell*; the shell, in turn may pass on the interrupt to its client, the editor.

4.3. Process Management Services

When a person at his terminal types a command, it is interpreted by his command interpreter and turned into a request to the appropriate service. Some services are always present and available, sometimes because they are vital to the operation of the system, sometimes just because they are heavily used. Other services, in contrast, are not always available, but have to be brought to life especially for the execution of a request. Usually, such services are not essential for the operation of the system and little used. Typical always-present services are the File Service of Chapter 6, the Bank Service of Chapter 7, and the

Process Service and the Boot Service, discussed in this chapter. Examples of services of the other kind which do not always have a server process waiting to carry out requests could be compilers, text formatters, mail service, etc.

Several services are responsible for enabling users to run their programs. First, there is the **Process Service** which handles all requests from clients to manipulate processes: it creates processes, it makes them execute; it can checkpoint, suspend and kill processes. Second, there is the **Pool Processor** service, which makes a *virtual processor* available for executing the program code. Then, there is the **Loader Service**, a specialised file server that stores programs. When a client wants to run a program, the Loader Service provides the code and data.

To clarify the relationship between these services, let us go through the typical scenario of creating and running a process. Process creation is initiated by a client sending a request to the Loader Server to obtain the **process descriptor** of the program to be run. The process descriptor contains information about the type of processor the program runs on, the memory requirements, whether it runs on any processor, or a particular dedicated one, and the process descriptor has three more capabilities that can be used to obtain the program's code, data and stack. The client can provide an environment for the process-to-be by replacing the *stack capability* by one that refers to an initial stack of its own. This provides a way of passing an environment to a new process.

The process descriptor is then given to one of the Process Servers as data in a request to create a new process. The Process Server allocates an appropriate processor, sends it the process descriptor and tells it to run the program. The elected processor—probably a Pool Processor—allocates memory for the process (it knows how much, since this information is in the process descriptor), and fetches the code, data and stack by sending requests on the capabilities enclosed with the process descriptor. The code and data capabilities refer to the Loader Service, and the stack capability can refer either to the client process, the client's Command Interpreter Server (or Shell), or to the Loader service.

The Process Server replies to the client with a capability for the newly created process. The client can then exert control over the process by sending commands on that capability. These commands are executed by the Process Server. For some more clarification we shall describe *checkpointing* a process.

Checkpointing is making a copy of the state of a running process, in order to be able to have a point for resuming execution of the process in case it should crash. Checkpointing can be ordered by the client owner of the process, but it is more likely that it will be ordered by the process itself, since it 'knows' the points at which it is safe to resume execution. The mechanism works as follows: The Process Server sends a command to the executing processor to stop the executing process and generate a process descriptor for it. For this purpose, the format of the process descriptor allows recording the state of an active process. Note that this is relatively easy, because the only interaction of a process with the outside world is through transactions. The process descriptor can be given to the Loader Server, which, can then fetch and store the process' code, data and stack.

At a later moment, the process can be resumed as if it were a new process.

Other operations on processes are *forking*, which is like checkpointing and continuing execution of both the original process and the copy; *migration*, which amounts to stopping a process, moving the process descriptor, and the code, data and stack to another machine where the process continues execution; *remote debugging*, which can partly be done by stopping a process, examining the process descriptor and memory contents, possibly modifying the state of the process and continuing execution.

The Pool Processors accept code from the Process Service and execute that code. These processors are part of the **Pool Processor Service**. This service has been kept as simple as possible so it can be realised in a microcomputer's ROM. Multi-programmed hosts can realise as many (virtual) Pool Processors as there are slots in their process tables.

For permanent services the **Boot Service** is available to help maintain them. Foremost, the Boot Service brings up the permanent services (File Service, Process Service, etc.) when the system comes up. This explains the name *Boot(strapping) Service*. The other task of the Boot Service is to take care of crashed server processes: A typical service, File Service, for instance, consists of a number of server processes. Due to software failures, hardware failures, power failures, etc., these processes can crash. The Boot Service helps client services by periodically checking if its server processes still work, and, if not, the Boot Service replaces them. Although this does not relieve the designer of a service from thinking about and implementing crash recovery procedures, it does make crash recovery simpler. Naturally, the Boot Service is its own client: when one of the Boot Servers crashes, the others replace it.

The next sections describe each of these services in detail.

4.3.1. Processes

A process (or cluster) communicates with other processes at different levels of abstraction: The highest level of concern to us is the level at which the process' code communicates with the outside world. A server process, for instance, receives requests and sends replies, using ports (and capabilities) that it, or its clients choose; let us call this level the **process level**.

One level down, however, there is another type of communication with a process: requests are used to control a process, such as requests to terminate, suspend or checkpoint it. We shall call this the **control level**. The control level is the level at which the owner of a process exerts control over it by making transactions with the process' managing service, the Process Service. Requests to terminate a process are interpreted and carried out by the operating system, but when a process migrates from one host operating system to another, these requests must be carried out by the new host operating system. In our implementation, a process carries a capability with it, created when the process was created, and given to the client that ordered the process' creation. This capability, the **control capability**, travels around with a process, and at any time, the operating system in charge of the process accepts requests through that capability to terminate, suspend, checkpoint, etc. that process.

At the lowest level, the **management level**, a process is viewed as an object, that is manipulated by its host operating system. This is the level of communication used internally by the services that provide process execution. The Amoeba Kernel, for instance, can be told to unload whatever 'code' resides in its memory, or to start (resume) or stop execution on the code in its memory. At this level a process is treated as a collection of instructions and bytes of data. The management level requests are the Pool Processor requests, which we shall discuss shortly.

As we have just seen we can look at a process at three levels, or, if one wishes, from three viewpoints: a process as a client or server, a process as an object from the 'client point of view,' from the outside, or as an object seen from the 'server point of view,' from the inside. In a sense ***Amoeba*** is *self-referential*: services define and implement objects, but services themselves consist of processes which, in turn, are also objects under control of a service.[27] However, there is no infinite regression here: the recursion bottoms out at the level of the physical processors of the system; they are not objects being created and deleted by some service, but offer service without needing another service to be able to exist.

The Process Level

At the process level of looking at processes we are concerned with the programmer's point of view. A process is governed by a program which describes the sequence of actions of the process.

The system calls provided by the Amoeba Kernel are rather different from the usual set of system calls that we know from most operating systems. To begin with, there are fewer system calls. The ***Amoeba*** operating system kernel does not provide file service, terminal I/O, time-of-day service, and all those other services present in many systems that we know. Instead, ***Amoeba*** provides only system calls making transactions. Using transactions directed to the Amoeba kernel, other system calls can be realised in an indirect fashion. In this way, no difference exists between using services provided directly by the kernel, and services provided by other processes, possibly on different hosts.

The Control Level

The control level describes the management of processes by other processes: the way a process can be created, and the way it can be influenced as it runs. Process creation and management resembles creation and management of most objects in ***Amoeba***: a request is sent to the manager of the objects—in this case the Process Service—to create a process. The Process Service returns a capability for the process, which has to be used in future requests concerning the process.

Process creation works as follows: A request must be sent to the Process Service, asking it to create a process. The request contains a *process descriptor* which describes the program to be run. The **Loader Service**, which will later be described in detail, is a specialised kind of file server that, given a capability for a program (which is viewed as a process in *statu nascendi*), can deliver a process descriptor for it, which, in turn, contains capabilities for the code and data of the program. The Process Service uses the information in the process descriptor to allocate a suitable processor and causes the process to be loaded and run. The client that created the process receives a **control capability** that it can use to exercise control over the process.

Here are some typical commands that can be given to control a process: One command is the command to *kill* the process. It stops the execution of the process, and clears up the remains. Another command is the command to *checkpoint* a process. This command causes the Process Service to save the current state of the process (by giving it to the Loader Service), and to return a capability for the saved state. This capability can later be used to continue execution of the checkpointed process as if it were a new process. Checkpointing can serve two purposes, namely to obtain a dump of the process' state to examine it, or to have a process image that can later be given to the Process Service to be run, if, for instance, the original process crashed. The checkpointed process continues to run as if nothing had happened. Naturally, a process can also checkpoint itself.

Killing and checkpointing can be combined in the *suspend* command: it saves its state so it can be given to the Process Service later to continue where it left off, and then kills the process.

There are also commands to give the current status of a running process and commands for debugging, such as commands to single step through a program, to set and clear breakpoints, etc. More about these commands when we come to the Process Service.

The Management Level

This section deals with processes as objects, mainly as seen from the inside.

The **state** of a process is all the information necessary for its execution. Among this information is the contents of the process' memory, its registers, its program counter, stack pointer, and the status of any system calls in progress. But this is not all: A process that executes on one processor cannot execute on every other processor. The CPU type for which the process was made, is also a part of the state. The next section is devoted to the representation of the *state* of processes.

The process state, just mentioned, exists in tangible form only when a process is transferred from one host to another, or when it is (temporarily) stored on a disk. When the process runs, its state changes continuously, and its representation is embedded in the state of the processor executing the process. When a process stops execution for one of a number of reasons, its processor gathers all state information into a representation of the process' state that is as machine independent as possible.

The representation of a process' state mainly resides in its **Process Descriptor**. The Process Descriptor tells on what CPU type a process must execute. It also tells the value of its program counter, its registers and it gives the state of various active system calls. As the Process Descriptor is reasonably small, it can be sent from one processor to another easily. The state of a process' memory is not included in the Process Descriptor, but three capabilities for obtaining the memory contents are included, one for the code, one for the data, and one for the stack. With these capabilities the memory contents can be read as if from a file.

Processes have the possibility of **migration** in ***Amoeba***; that is, a process that has been executing for a while on one processor can be moved to another to continue its execution there. Migrating a process is almost identical to loading one; in the former case, a process moves from one processor to another, in the latter it moves from a Loader Server to a processor. Checkpointing can be seen as a special form of migration: a process then moves from a processor to a Loader server. A combination of checkpointing or suspending a process with looking at, and possibly changing its saved state can be a powerful debugging tool, especially if it is combined with a processor that allows setting breakpoints.

An important property of this model is that initially loading a process into a processor is no different from migrating a process into a processor, so even on systems where the Process Service does not make use of migration for scheduling, but processes are assigned to one host for their duration, clients can be allowed to have much more control over processes than in most operating systems (checkpointing, suspending, modifying processes).

4.3.2. The Process Descriptor

The Process Descriptor describes the process' state, the contents of its registers, the state of system calls in execution, it contains accounting and scheduling information, and two capabilities, the process control capability and the code capability for obtaining the memory contents of the process. Before we discuss each of these items we refer to FIGURE 4.3 where, globally, the lay out of the Process Descriptor is given.

The *control capability* is the capability giving the right to manipulate the process at the control level (start execution, stop execution, give process status, etc.). The processor executing the process receives the control commands for this capability and carries them out.

The *code*, *data* and *stack capabilities* give access to the process' code, stack, and data, respectively. When a process descriptor is made (this happens when a process is created, checkpointed or suspended), a code capability is made that can be used to fetch the code and data associated with the process. The Process Descriptor containing the code capability can move from one host (processor, Loader Server) to another, and code and data can be fetched by the latter, using the code capability.

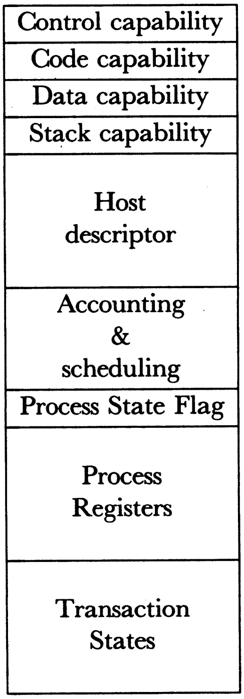


FIGURE 4.3. *Amoeba* Process Descriptor lay out.

The *host descriptor* is a list, where each entry describes some aspect of the host that the process must run on. Each entry consists of a one-byte *length*, a one-byte *type*, and zero or more *argument* bytes. The terminating entry has a zero *length* field. Each *type* indicates a different aspect required of the host processor. FIGURE 4.4 gives a list of some of the types and their meaning.

The *accounting & scheduling* entry can be used by a processor to convey useful scheduling information and amounts of used resources to a client Process Server.

The *Process State Flag* gives the 'overall state' of the process: *running*, *pausing*, *suspended*, *exited*, *killed* or *crashed*. The last three states are practically the same; they exist to inform the owner of the process of the cause of the process' demise.

The *transaction states* give the states of outstanding transactions: request and reply ports, buffer addresses and lengths and the state of the transactions. Timers and retransmission counts can safely be (re)started from zero.

type	argument	description
HOST	capability	Specifies the host processor for the process. Dedicated processes can thus contain a capability for running on some specified processor or class of processors.
CPU	type	Indicates the CPU type (instruction set) the process uses.
CPU_SUPPORT	bit_map	The CPU-dependent bit map indicates which extra features the process needs in its host (*e.g.*, floating point, memory management, commercial instruction set).
MEMORY	size	The (maximum) amount of memory needed by the process.

FIGURE 4.4. An overview of some of the host descriptor entry types.

4.3.3. Processors

Processors form the heart of a computer system. This section describes the different kinds of processor service in ***Amoeba***, private processors, dedicated processors and *pool processors*. The private processors are of no concern to us here; they are processors, connected to the ***Amoeba*** network so they can use ***Amoeba*** services, but they are not under control of an ***Amoeba*** service. Their owners decide which programs they will execute.

Before discussing the details of processor service, it is important to explain what we define processors to be. A **physical processor** is the hardware necessary to execute programs, usually a CPU and memory, plus, sometimes, peripherals. A **virtual processor** is an environment in which a process can execute. In general, each physical processor provides an environment for the execution of at least one process, usually the operating system. But an operating system itself also provides an execution environment for processes, user processes. It realises a number of virtual processors. Some physical processors with their operating system implement one virtual processor; we call these monoprogrammed. Other physical processors have an operating system that implements several virtual processors; those are multiprogrammed. Some operating systems can be user processes running in a virtual processor (*i.e.*, on top of another operating system). One virtual processor can thus split itself into many, if the *virtual operating system* is a multiprogramming system. In the remainder of this chapter we shall use the word processor to mean a virtual processor.

In ***Amoeba*** there are several classes of processes that need to execute: dedicated processes, very long-lived server processes, short-lived server processes, and *'user'* processes. The word *'user'* is in quotes, because in ***Amoeba*** the distinction between system processes and user processes is one of viewpoint. The 'system' makes no distinction. Dedicated processes are processes that must run on a particular physical or virtual processor. Examples of dedicated processes are operating systems, processes that control devices, such as disks, phototypesetters, etc. Server processes accept requests from clients, carry them out, and return replies. Some servers are long-lived; that is, the server process is started up, and then supposed to run for ever. If the process crashes it has to be replaced. Short-lived services are usually created to carry out one request, and then removed again. These services are typically infrequently used services for which a dedicated processor is a waste of resources. The remaining processes are the familiar 'ordinary' processes, user processes which usually run once.

4.3.4. Dedicated Processors

The dedicated processes are the easiest to manage. They always run on the same physical processor, and the only management activity associated with these dedicated processes is to check at regular intervals to see if they still operate, and, if not, to attempt to start them up again.

Most physical processors, when they are switched on, automatically run a bootstrap program in ROM. If they have a local disk, the bootstrap program can load its host's operating system, the operating system can load the server process, and off the server goes. Not every physical processor has a local disk, however. In such cases, the dedicated processes must be loaded through the network, which is straightforward, assuming there is a service in the network that is able and willing to provide the binary.

The service that provides process images for its clients is the **Loader Service**. It is implemented by one or more dedicated servers that each have a local disk. The Loader Servers can easily bootstrap themselves from this disk. As far as dedicated services are concerned, the Loader Service provides a very simple file service. The bootstrap program in the dedicated processor's ROM (or the virtual processor's operating system) sends a request to the Loader Service to give it the image of the process it wants to run, and the Loader Service provides it. We shall discuss the Loader Service in more detail in § 4.4.

It is an interesting observation that inside the allocable Pool Processors are dedicated processors; the dedicated process executed by them is the operating system, which implements one or several virtual Pool Processors. The next section describes Pool Processor Service in some detail.

4.3.5. Pool Processors

The allocable processors accept work from their clients, usually the Boot Server or the Process Server. The service, offered by the allocable processors to the system is the **Pool Processor Service**, which we shall now describe.

Pool Processors are virtual processors; each Fool Processor handles at most one process at a time. Roughly, the service works like this: A client (Process Server or Boot server) sends a request to a Pool Processor to allocate it, load it, and execute the loaded code. When the loaded process terminates (either naturally, through a crash, or by being stopped by the client owner), a Process Descriptor is made for the loaded process, and the process' memory contents are made available. The Pool Processor can then be deallocated after which it is ready for the next client.

The set of requests and replies defined by the Pool Process Service is quite simple and straightforward. Typically, a client uses a Pool Processor as follows: First it sends a request to it to allocate the Pool Processor. This request contains a process descriptor for the process to be run. While the Pool Processor is allocated it will not accept requests to do work for other users. The Pool Processor then fetches code, data and stack using the capabilities in the process descriptor, and returns a capability for managing the process to the client. The client can then order execution to be started, stopped, checkpointed, migrated, etc. Finally, the client deallocates the Pool Processor, allowing it to accept work from other clients.

Ordinarily, regular user processes will not be allowed to use Pool Processor Service directly; that is, most users will not have the appropriate capabilities to access a Pool Processor. If a user wishes to run a program, he must send a request to a 'Process Server', which checks budgets, quota and permissions, builds up the process, and finds a Pool Processor to do the job.

A sketch of a list of requests, accepted by the Pool Processors follows. *Pcap* represents the capability needed to access the process in the Pool Processor. *Alloccap* is the capability needed for allocating a Pool Processor. Pool Processors only listen to this capability when available.

allocate(*alloccap*, *processdescriptor*)

Allocate and load a Pool Processor. A capability is returned for controlling the loaded process.

start(*pcap*)

Start a loaded process.

stop(*pcap*)

Stop a running process.

status(*pcap*)

Returns the state of the executing process.

unload(*pcap*)

Build up and return a process descriptor for the current process. Allow the code, data and stack to be fetched.

deallocate(*pcap*)

De-allocate the Pool Processor. Allow it to accept another process.

report event(*pcap*, *events*)

Reply to this request when one of the specified events occurs. Typical events are: process terminated; process started to pause; process ended pause.

Some commands may be bundled into one. For instance, *allocate* and *start* will often be done in one blow. The same applies to *stop* and *unload*, or *stop* and *deallocate*.

Pool Processors, capable of tracing, single stepping, and what-not can accept a few more calls, useful for program debugging. A Debugging Server could use such Pool Processors to provide to ***Amoeba*** users a powerful debugging tool.

Pool Processors provide service to their clients, but unlike most services, they do not usually receive a request to do some 'bounded' amount of work; in a Pool Processor a request (*e.g.*, *start*) may cause a Pool Processor to run for ever. This does not matter normally, because its client (*e.g.*, the Process Service) will stop the Pool Processor if the process runs too long. But, hardware and software being what they are, processes do not run forever, and even a Process Server can crash. All the Pool Processors allocated to a crashed Process Server must somehow be returned to the pool of available Pool Processors. Some of them may find out by themselves that their client has crashed, because they will attempt to send a reply and fail, but others, waiting for the next request, will not find out and wait forever.

We have considered two solutions to this problem. The first is to build timers into the Pool Processors, which, if no requests come in during the timer interval, automatically stop any running processes and return the Pool Processors to state *free*. The other solution is to let the Boot Service—which keeps track of the Process Service anyway and detects Process Server crashes—find 'orphaned' Pool Processors and return them to the free pool.

We have chosen the second solution. The first solution is not elegant, because it requires both the Pool Processors and their clients to maintain timers. These timers must be shorter than the Pool Processor timers, and when these expire the client must make some sign of life to its Pool Processors. The Pool Processor's timeout time must be globally known and can not simply be changed, and the client's timers must be shorter by some unknown amount that depends on message transmission times and network and operating system delays. In the other solution, the Boot Server plays an important role, because it has to find the orphaned Pool Processors. This does not, however, greatly increase the complexity of the Boot Server, since it has to manage some of the Pool Processors anyway; the Boot Server uses Pool Processors to execute (some of) the server processes under its management.

4.3.6. Migration

The process of migrating a process from one host to another is tricky. It has to be done carefully so as not to disturb outstanding transactions (*trans* or *getreq*). A process, while being moved from one processor to another cannot execute instructions, but its state can change because transactions complete or requests arrive. Received messages especially must be treated right: Imagine, for instance, a process being migrated because it had been blocked waiting for a message. Another process may send it a message (request or reply) while it is being moved. If this message is not received, nor its retransmissions, the sending process decides incorrectly that the process being migrated has died. This should not happen.

An additional problem may be caused by outstanding transactions with the local operating system. The operating system kernel must either delay migration until the request has been carried out, or send the reply to the process' new host. The choice depends on the kind of request made to the kernel.

Not every process can be migrated. A process, for example, waiting for a disk interrupt (reply from a request to the disk) cannot be migrated to another processor (with another disk) to do the next transaction with the disk there.

Let us assume process P has to migrate from host A to host B. First host A must suspend the process. Then a Process Descriptor can be made, describing P's state. The Process Descriptor is then sent to host B, which, upon reception, immediately does P's outstanding *get*s, according to the state of the outstanding transactions. Retransmission timers and piggyback timers can all be restarted from scratch. The process is deaf from the time a process descriptor is created at host A until the time host B receives the Process Descriptor. This time is, approximately, one message time plus the overhead of making (at host A) and interpreting (at host B) the Process Descriptor. This time can be kept small, so the probability of message loss can also be kept small. However, sometimes, messages will be lost. But this is acceptable, because the transaction protocol was designed to recover from lost messages.

When host B receives P's Process Descriptor, restarts timers for it, and does its *get*s again, migration is by no means complete. Host B must first use the *code*, *data* and *stack capabilities* in the Process Descriptor to obtain P's memory contents. A little care has to be taken by B to restore P's memory contents 'around' any messages received in the mean time. When B has completed the migration, A can remove P's Process Descriptor and free the memory, occupied by it. B can start execution of P.

As a variation on 'ordinary' migration, we can also consider *'swapping.'* Swapping, in a multiprogrammed machine, means temporarily writing a process' state to secondary storage to make room for another process. In an ***Amoeba*** environment, swapping is temporarily migrating a process to a special *swap server*, a host with secondary storage that can store the state of many processes at a time. The swap server is not just a specialised file server, however, because the swap server has to be able to handle a swapped process' outstanding *get*s and timers. This requires some knowledge about processes' internal structures—which differ for processes for different target machines.

Another special case of migration is the initial loading of a process. The Process Service gives a Process Descriptor to a Pool Processor, which then uses the code capability to retrieve the process' code and data from the Loader Service. The Pool Processor uses exactly the same protocol for downloading a migrating process, for swapping in a process, and for loading and starting a new process.

4.4. Loader Service

The **Loader Service** has already been mentioned as a service that provides process images to its clients. Basically, it is a simple service that responds to commands to store and retrieve processes. A process—as shown before—consists of a Process Descriptor plus code, data and stack. When the capability of a process is given to a Loader Server, it responds by returning the Process Descriptor. The Process Descriptor contains capabilities for code, data and stack, and with these capabilities the process' code, data and stack can be read as if from a file.

Programs or processes are easily stored by the Loader Service; all a client must do is request it to store a process and give it the Process Descriptor for it. The Loader Service stores the Process Descriptor, uses its code capability to obtain and store the program's or process' codes and data, and returns a capability for the process.

The protocol that the Loader Service uses for moving processes between hosts is the same all over: first the Process Descriptor changes hands, then the code, data and stack capabilities are used to transfer the rest of the process. The importance of a uniform procedure for process transfer is, of course, to hide the difference between process downloading, swapping, migrating, checkpointing, dumping, etc. A Pool Processor need never know whether it receives a process from another Pool Processor, from a Swap Server, a Process Server, or a Loader Server, nor need it know where it sends a process.

The basic functions of the Loader Service can be augmented by some extra service in heterogeneous systems. A request for a Process Descriptor can be accompanied by a description of the host the process must run on. An intelligent implementation of the Loader Service could store several versions of a program, one for each of a set of processors. It could then deliver several Process Descriptors and several code and data images, as required by its clients.

Such service can be implemented by storing processes in some machine independent code, that can be quickly expanded into the target code of the desired processor. This should be combined with a cache for target machine versions of programs, because code generation from an UNCOL* is too slow to do it on every execution of a program.

An even further refined version of the Loader Service could store program sources and compile into desired target code upon request. Again, this should be combined with a cache of frequently used binaries, because of the speed of current compilers. Such an integrated compiler/loader/program store no longer deserves the name Loader Service, but perhaps *'Program Service.'* The technology

* An UNCOL (UNiversal Computer Oriented Language) is an intermediate language, resembling machine language, for which compilers produce code.[31] The UNCOL code is then translated into target machine code. With this scheme it is possible to get compilers for n high-level languages on m target machines by writing n compilers and m translators. Without an UNCOL, $n \times m$ compilers would be required. The ***Amoeba*** system, which runs on various CPUS, uses the Amsterdam Compiler Kit, based on an UNCOL.[74]

for making such a system is known[74]; it is an interesting project to build it to test its practicality.

At a glance, it would seem that Swap Service and Loader Service are very similar: both services store processes, although the Swap Server stores them only for short periods of time. There is an essential difference, however, between the two: The Swap Server stores active processes, whereas the Loader Service stores programs, *disguised* as (inactive) processes. While a process is stored by the Swap Server, it does not execute instructions, but it is still active: its transaction timers continue to run and may well expire while the process is stored on the Swap Server's disk; its outstanding *get*s may still complete by the reception of a message.

4.5. Boot Service

Important goals of distributed system design are reliability and availability. The prime mechanisms to achieve these goals are replication and fault tolerance. Every service, if it is to be reliable, must be designed in a way to expect errors in subservices, but it must also be designed to expect that it will occasionally crash itself. As an aid in recovering from crashes the **Boot Service** has been designed. However, it is not more than just an aid. No service can be implemented to complete error recovery for any other service. Services must do their own crash recovery, but the Boot Service can help to detect crashes, and create server processes to replace crashed ones.

Boot Service must be a distributed service itself. If there were one Boot Server process in the system, it could crash, and Boot Service would no longer be. Boot Service consists of several processes which keep an eye on each other. If one server crashes, another is created to take its place. In this way is continued Boot Service guaranteed in the face of single-site crashes.

The Boot Service has to do three things: keep itself going, keep client services going, and put orphaned Pool Processors back in their *free* state. We shall now examine each of these tasks in turn.

The Boot Servers have to check periodically if their colleague Boot Servers are still functioning. If it is detected that one of them has crashed, it is restored by one of the others. Crash detection and restoration must be done in an orderly way; for instance, when two Boot Servers simultaneously detect that a third has crashed, they might both create a new one. Synchronisation techniques for this problem are well known.[17, 38, 56] An appropriate one can be selected, depending on the number of Boot server processes, the type of the network and other parameters.

Each of the Boot Servers receives polls from its colleagues on a unique port. Besides its own unique port, every Boot Server must know the ports of the Boot Servers it polls, plus enough information to restart them when they crash. In both models this implies that every Boot Server must know the ports of all the other Boot Servers.

A service that the Boot Service keeps an eye on generally consists of a number of server processes. These processes need not all be the same, but could be different processes, each specialised to offer some part of the service. The number and kind of server processes must be dynamically changeable, so it is possible to create more server processes for a service when it is heavily used, and reduce the number of server processes when business is slack. Per client service, the Boot Service maintains two tables, one that gives the desired state of the client service (in terms of which processes should run where), and one that gives the actual state of the client service, which may not be completely up-to-date.

Information on a process that should be running is called a **template**. Information on a running process is called an **instance**. Normally, there is a one-to-one mapping of templates and instances, but occasionally a crash destroys a service process which temporarily disrupts the one-to-one correspondence.

The *template table* holds the following information:

An *instance pointer*, referring to the corresponding entry in the *instance table*. When there is no running service process for the template, this pointer will be set to nil.

A capability for the program to be executed as the server process. When the server process has to be booted, or rebooted, this capability is used to find the program to start.

The *allocate port* for the server process. This is a port which gives access to a processor where the server process can be run. This can be a Pool Processor, or a dedicated processor; in the former case, many processors will offer service on that port, in the latter case, only one processor will listen on the allocate port.

A *test request* and *reply*, to be used to test whether the corresponding instance of the server template functions. The *test request* can be a request to the server process to carry out a self-test operation. If the correct reply is not given or if no reply is given at all, the Boot Server will assume the tested instance has crashed.

The template information is 'semi-permanent.' It survives crashes of both the service it refers to and the Boot Service. The template information for a service is given to the Boot Service when the service becomes a client of the Boot Service. It can be changed if the number of servers for that service should change, or if something else about the service changes.

The *instance table* changes much more rapidly. Whenever a server process crashes, or a new service process is created, this is reflected in the instance table. The instance table contains:

A *template pointer* which refers to the template on which the server process is based.

The *server capability*, the capability used to address the test request.

The process capability, with which the server process is controlled.

Templates can be added and removed dynamically, increasing or decreasing the number of server processes. When a new template is added, the Boot Server creates a new server process for it; when a template is removed, the Boot Service's client can specify if he wants the associated server process to be killed, or if he just wants the Boot Service not to create a new server process when the old one terminates.

Note that bootstrapping is just a special case of the normal Boot Service task: all processes crashed. The Boot Service needs non-volatile storage for the templates, so when Amoeba is bootstrapped by starting the Boot Service, the Boot Service knows what processes to create. At least one Boot Server needs a local disk where the templates are stored; it must be a local disk, because regular Disk Service cannot be assumed to be operational when the system is down.

When bootstrapping the system, some care has to be taken about the order in which various services are started up; services form a hierarchy, so some services depend on the availability of others to function. On becoming a client of the Boot Service, a service could specify on which subservices it depends. The Boot Service can then generate a dependency graph. If it contains no cycles, it is a simple matter for the Boot Service to start the services in the correct order. If it does contain cycles, the Boot Service can attempt to start all services in the cycle simultaneously, or as closely together as possible.

Pool Processor service can generally be found right at the bottom of the service hierarchy. Its availability is therefore essential for the operation of most services, including Boot Service itself! Pool Processors cannot generally be brought up the same way other services are brought up; when a Pool Processor crashes, this usually means the processor on which the Pool Processor runs has to be reset. If the network provides mechanisms for resetting processors, the Boot Service can do this. Otherwise it will have to be done by hand.

5

PROCESS SCHEDULING

In most operating systems, when a process is run, it is assigned to a processor for execution. In time-sharing systems processes are assigned for short periods, tenths of seconds, and usually spend the time in between waiting for I/O to complete, or for a new time slice in the processor. In a large distributed operating system, with hundreds of processors, processes may be exclusively assigned to a processor for the process' life time. The decision making process of when to allocate which processes to which processors is called **scheduling**. This section presents a scheduling algorithm suitable in a number of different environments based on pre-emption and notions of fairness, maximising throughput and minimising response times.

In traditional operating systems, there was just one processor that had to serve many users. Multiprogramming, or time sharing was the technique usually employed for such systems. Hardware prices are still dropping, especially those of processors and memory; distributed systems will soon have tens, hundreds, perhaps thousands, of processors. On such systems, with current computer use, every process can easily be assigned to a private processor of its own for its life time. However, as our computing environment changes, the demands we make on it will also change; as more processing power becomes available, we shall also need more of it. Our programs will become bigger, more complex, intelligent, and we shall try to exploit the possibilities of parallel computing. However many processors we have, we shall never have enough.

Programs, as they are written today, spend most of their time waiting for input or output. Some programs, of course, are intended for interactive use, which causes them to be idle nearly all the time, waiting for human users to think up and type their input. Other programs also have to do a lot of waiting: waiting for disk I/O to complete, waiting for confirmation that a data base update has been done, waiting for a tape to rewind, waiting for another process to feed more data into it, etc. Simple measurements in the UNIX operating system have shown that processes spend six times as much time waiting for system calls to complete as time computing. In theory, by removing a waiting process from its processor, and replacing it with a running process, throughput can be increased by a factor six.

Changing the processor allocation is called **process switching**. In different environments process switching occurs in different ways: In a centralised multiprogramming system a switch consists of changing the memory map, some registers and setting up a new environment. Some computers have a single instruction to do this; in others it takes a few milliseconds. In a distributed environment, such as ***Amoeba***, a process switch can be used to balance the load on a number of processes; the switch could then consist of sending the state of a running process from one processor to another. Such a switch is a much more time-consuming affair. In the next sections we shall not go into the details of how switching is done, but rather into the decision making process of what switches must be made and when.

Optimal load balancing between the processors of a distributed system can only be achieved with global and up-to-date knowledge of the state and load of each processor and the behaviour of each process. The nature of distributed systems makes it impossible, however, to have totally up-to-date information about the state of the system at all times. Furthermore, the behaviour of most processes that run in an open, general-purpose system is not usually known in advance.

In all operating systems, scheduling decisions must be made based on imperfect information. A process' future behaviour has to be predicted using statistical information about processes in general and extrapolations from processes' past behaviour. In distributed systems these predictions are necessarily less accurate than those that can be made in conventional, centralised, closed operating systems, in which the scheduler has more information about what the process is doing than in distributed systems. For instance, when a process is waiting on an event, the scheduler in a traditional operating system usually knows what event, so it can estimate quite accurately when the event is likely to occur: A process waiting for disk I/O to complete will probably be blocked for a short time, while a process waiting for terminal input is likely to be blocked much longer. In ***Amoeba*** this information is not available. ***Amoeba*** is an open system, where all requests are made through the network. The scheduler cannot look at these requests, so it cannot know when a reply is due; it can only estimate from past experience.

As an illustration of making predictions based on a process' past behaviour, consider a process copying a large file. The process will alternately make read requests and write requests to a File Server. The Scheduler can observe the time the process is blocked waiting for the reply, and predict, when the process waits next, that the delay is approximately going to be the same as in the past. Such an estimate will usually be correct, although it is possible that every now and then it will be widely off the mark when, for instance, a process suddenly decides to request terminal input instead of doing disk I/O.

Deterministic scheduling algorithms are well-known.[10] In these algorithms, advance knowledge is used of processes' execution times. Generally, however, such knowledge is not available, and scheduling algortithms must be used that do not use advance knowledge of process run times, or that use *predictions* of process execution times. Scheduling algorithms that do not use such estimates are used in most operating systems. Typical algorithms are: *round robin, shortest elapsed time,* and *priority queues.* An algorithm that requires advance knowledge of process' run times is *shortest remaining run time.* As the name suggests, this algorithm schedules the process with the least remaining run time at each moment. For this algorithm, the mean number of processes in the system is minimal,[43] which is a desirable property for an operating system.

A *shortest remaining run time* algorithm for scheduling must know about the remaining run time of the processes is schedules, or at least be able to estimate it. Generally speaking, remaining run times are just as unpredictable as wait times. Several ways exist, however, to make better estimates than, for instance, the average run time of every process executed in the past. One way is, to gather run-time statistics for specific programs that are executed regularly, to use these for a better prediction of future run times. Note, by the way, that run times of most processes depend heavily on the size and complexity of the input (unseen by the Process Server), causing considerable variation in run times.

5.1. Runs and Pauses

Before going into the details of scheduling algorithms it is necessary to know a little about the behaviour of processes. Typically, a process executes instructions, makes system calls, executes more instructions, makes more system calls, and finally terminates. Usually, making a system call involves disk accesses, terminal input, tape movement or other I/O. The process then blocks, until the system call is completed. During this time the process cannot execute instructions, and, usually, the operating system makes a process switch to execute a process that is not blocked. Both in traditional time-sharing systems and in other distributed systems, processes alternate between periods of executing instructions, and periods of waiting for external events.

It is important to estimate the remaining time a process will execute instructions before it pauses again, because this information is needed to decide which process to allocate, and for how long it is likely to remain allocated. A period of executing instructions between pauses is a **run**, and we shall now attempt to find methods to predict the lengths of runs.

In order to determine which times exactly have to be estimated let us examine the hypothetical path of a process through the system of FIGURE 5.1. We see a process being created at $t0$. The scheduler allocates it to a processor at $t1$. Then the process runs for a while, until it blocks (waiting for I/O) at $t2$. The

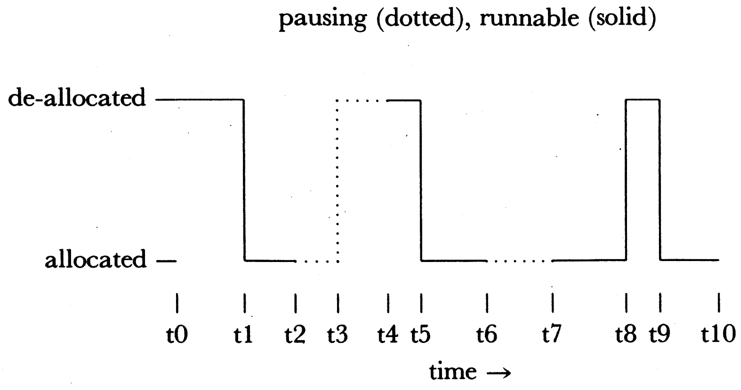


FIGURE 5.1. The path of a hypothetical process being scheduled in an operating system.

scheduler decides to remove the process from that processor temporarily at *t* 3, probably in order to use the processor to execute another process. The event that the process was waiting for occurs at *t* 4 and the scheduler finds a processor again at *t* 5. This story continues until *t* 10, when the process terminates.

Let us define some terminology to aid discussion. The execution of a process consists of a sequence of **runs** and **pauses**. A run lasts from the moment a process becomes runnable to the moment it terminates or pauses to wait on some event. The pause that then starts lasts until an event occurs that makes the process runnable once more. For scheduling, we are interested in the lengths of runs and pauses, but before we discuss run times and pause times, we must note an important difference between the two. When a running process is suspended, it continues to be runnable, but no instructions are being executed. A run must therefore be divided into periods where a process is *running* and periods where it is merely *runnable*; that is, it is not executing instructions, but it could be if only it had a processor to do it on. Together, the durations of these periods yield the *real time* of a run. However, for each run we are primarily interested in the sum of the periods where a process is running (runnable and allocated), the *run time*. Note that in a run, the run time plus the suspended time equals the real time of the run.

Pauses are different. The event that terminates a pause will occur independent of whether the process is suspended or not. The *pause time*, therefore, is the time between the start and the end of a pause, and the duration of a pause is not influenced by how the pausing process is scheduled (although it may depend on how other processes that must eventually cause the event that terminates the pause are scheduled).

Looking at FIGURE 5.1 again, we see that our hypothetical process has three runs ($t0$ to $t2$, $t4$ to $t6$, and $t7$ to $t10$), during which it is suspended three times, once in each run ($t0$ to $t1$, $t4$ to $t5$, and $t8$ to $t9$). It has two pauses ($t2$ to $t4$, and $t6$ to $t7$).

The CPU time of a run is determined by the process (program), its input, and the speed of the processor that executes it. The scheduler has no influence on run times, but, in order to allocate processes to processors efficiently, schedulers need to know or to estimate them. Suspended times are under scheduler control. Scheduling algorithms will make an attempt (among others) to minimise these. Pause times, finally, are determined by external events, such as when a user at a terminal presses the return key, or when a disk request completes, or when a query sent to a data base server gets its reply.

We want to develop useful scheduling algorithms for the processes under control of the scheduler. These processes include short lived client or server processes that do their work and then terminate, but also very long lived server processes that execute a request, return a reply, and then wait for the next request to arrive. Potentially these processes live for ever and disappear only when they crash or when they are explicitly terminated.

Because of this mix of short lived, long lived and 'eternal' processes we do not use 'total' execution times of processes in our scheduler model, but base scheduling on the (predicted) lengths of individual runs and pauses.

In the next section we shall develop a model for predicting the length of runs and pauses, using no other information then the knowledge how long a run or pause has already run or paused.

5.2. Estimating run times and pause times.

When we wish to estimate how long a run or pause is going to be, we can use several sources of information to base such an estimate on. First, we can use statistical information about processes in general. Run times and pause times of some large collection of processes executed can be measured and collected, and their distribution can be computed. We have made such measurements, and below we shall present the results. Second, when a particular program (*e.g.*, Pascal compiler, text formatter) is executed very often, we can also gather the same statistics for that particular program, yielding a more accurate prediction of run and pause times for frequently run programs. (The Loader Service could store this data along with the code.) Third, we can use information that is gathered as the process runs to predict its future behaviour. We have already mentioned how we can observe, for instance, that a process makes a sequence of requests, followed by similar pause times. We can also safely assume that a process that has executed for an hour will probably not finish in the next few seconds. Finally, we can combine these methods to obtain even more accurate predictions.

We shall examine measurements made in our Unix system, and use these to derive a model for the distribution of run times. Given such a distribution function, the expected run time of a process can be computed. It is also possible to compute the expected *remaining* run time when the run time consumed so far is known.

Measurements were put in the operating system's code to register the moment processes were blocked and the moment processes became runnable again. Process creation was treated as a special case of a process becoming runnable; process termination was a special case of a process ceasing to be runnable. Both for runs and pauses 32 counters were installed, such that counter i ($i = 0, \cdots, 31$) counted the number of runs or pauses of length between $2^{i-1} - 1$ and 2^i clock ticks of 20 ms. From the probability density function thus obtained the mean run time and mean pause time can be computed. The mean run time was found to be 26.4 msec, and the mean pause time 160 msec. In Figure 5.2, a bar graph shows the measured probability-density functions for runs (top) and pauses (bottom).

A few notes are in order before any conclusions are drawn. Because of the logarithmic scales used in the figures the beholder may falsely put too much trust in the values in the rightmost columns. These represent samples of a size of on the order of 10^{-6} of the total number of samples: a few. One more sample in the rightmost column in the graph would already make a significant difference. Some of the columns in the histogram overlap. This was done to reflect the inaccuracy in the measurements: A process that received 1 clock interrupt during its run has had a run between 0 and 2 clock intervals; a process that received 2 interrupts has had a run of between 1 and 3 clock intervals. This explains the overlap.

A first glimpse at the measurements shows both a surprisingly large number of very short runs and a surprisingly small number of runs of a minute or more. Two causes can be put forward to explain this phenomenon. Firstly, the measurements show statistics of individual runs, not individual processes. Most processes are I/O bound, and do very little processing between being blocked on system calls. These processes execute many system calls, however, which have a significant effect on the size of the leftmost columns. Few processes are so CPU bound that they compute without executing a single system call for a solid minute. Those that do, however, make few system calls and have only little effect on the right-hand side of the histograms. Secondly, the measurements were made over a period of a few weeks on one computer science faculty computer. Most of the users were staff working on various projects, most of which involved program development and text processing. Just one user doing, for instance, a project to try to find a new largest known prime number, would have had a significant effect on our measurements. However, no such user was on the system during the time these statistics were gathered.

In order to develop a mathematical model for estimating run times, a probability density function $p(t)$ will be fitted to the measurements to give the

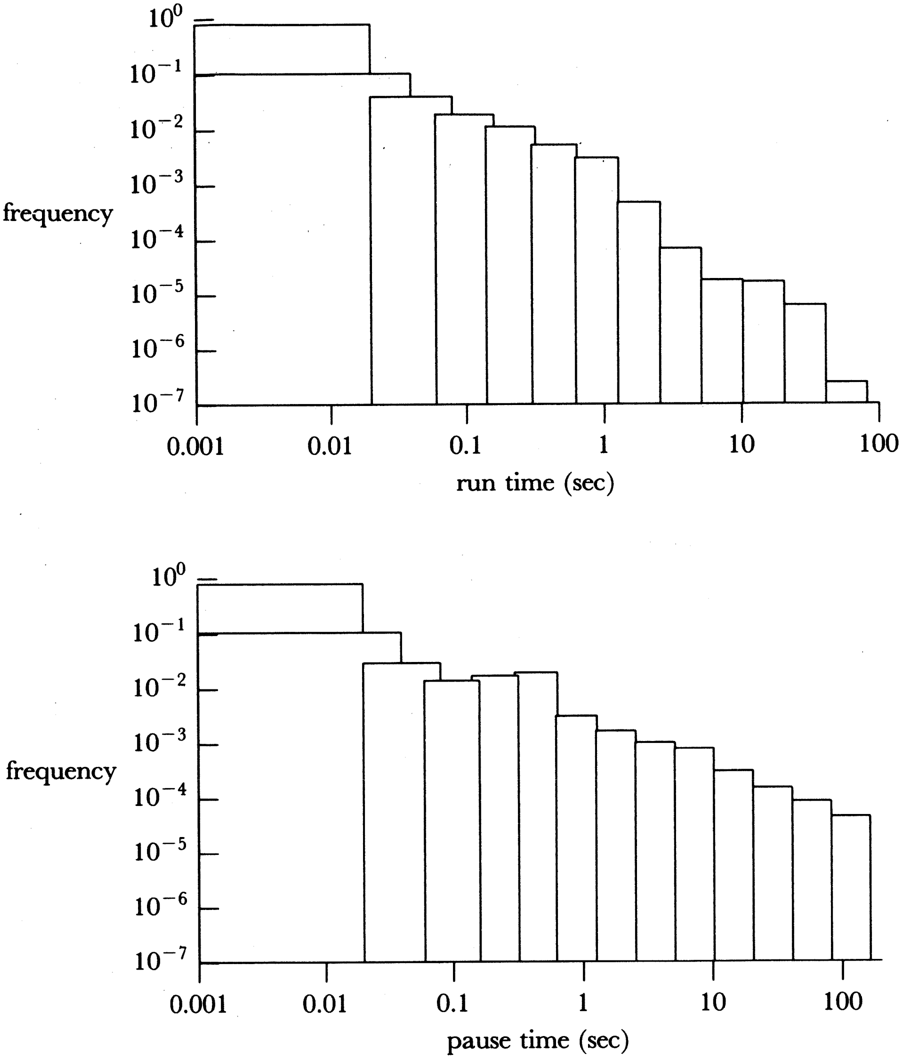


FIGURE 5.2. Run-time and pause-time distribution for the UNIX operating system. The bar graphs are based on measurement of slightly more than 20 million runs and pauses.

probability, $P(t1,t2)$, that a process starting a run at $t=0$ ends that spell between $t1$ and $t2$ by

$$P(t1,t2) = \int_{t1}^{t2} p(t)\,dt$$

$p(t)$ must be normalised, so

$$\int_0^\infty p(t)\,dt = 1$$

The expected run time is given by

$$\int_0^\infty t\,p(t)\,dt$$

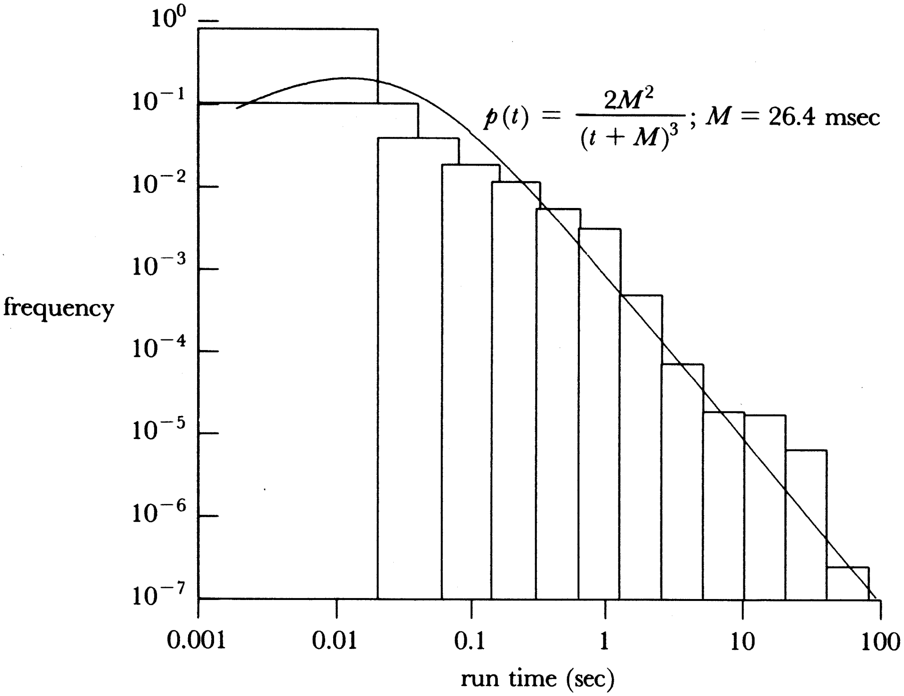


FIGURE 5.3. Measured run-time distribution with run-time distribution derived from $p(t)$ superimposed

A function that was found to fit the run-time measurements accurately was

$$p(t) = \frac{2M^2}{(t + M)^3} \tag{5.1}$$

as can be seen in FIGURE 5.3 where p is superimposed on the measured histogram (with M set to 26.4 msec). In order to examine the properties of this probability density function, let

$$\Phi(T) = \int_T^\infty p(t)\,dt, \text{ and } \Psi(T) = \int_T^\infty t\,p(t)\,dt.$$

Computing the integrals we get

$$\Phi(T) = \left[-\frac{M^2}{(t+M)^2}\right]_T^\infty = \frac{M^2}{(T+M)^2}$$

and

$$\Psi(T) = \left[-\frac{M^2 t}{(t+M)^2} - \frac{M^2}{(t+M)}\right]_T^\infty = \frac{2M^2T + M^3}{(T+M)^2}$$

We can easily see now that p is normalised: $\Phi(T) = 1$ for $T = 0$. The expected run time for a process is $\Psi(0) = M$. The expected run time of a process, given that it has already run for a time T is

$$\Upsilon(T) = \frac{\int_T^\infty t\,p(t)\,dt}{\int_T^\infty p(t)\,dt},$$

which is

$$= \frac{\Psi(T)}{\Phi(T)} = \frac{2M^2T + M^3}{M^2} = 2T + M, \qquad (5.2)$$

This simple model gives us an instrument for estimating the lengths of runs. When a run starts, we predict its length to be M, but as it lasts we revise our prediction, based on the knowledge that it has already last some time. It is interesting to observe the following: if we have exact or nearly exact information about run times, estimates of their lengths will not change or hardly change as it lasts; the estimate of the time between now and the time a run ends decreases as time elapses. In our model, however, it is exactly the other way around. As time goes on, the model predicts that the time between now and the moment a run ends increases. A process that has run for a time T is expected to run for a time $M + T$ more. In the next section we shall see that this has consequences for the scheduling algorithm.

The model for predicting run times can be further refined; if the Scheduler passes measured times to the Loader Service, the information is available when the process is run next. The Loader Service can collect the data and compute expected times and variations.

For pause times another interesting probability density function was found to fit the measurements:

$$p'(t) = \frac{2M}{(t+M)^2}$$

This probability density function has an expected pause time of ∞! Yet, as can be seen in FIGURE 5.4, it yields an good approximation of the measured pause times.

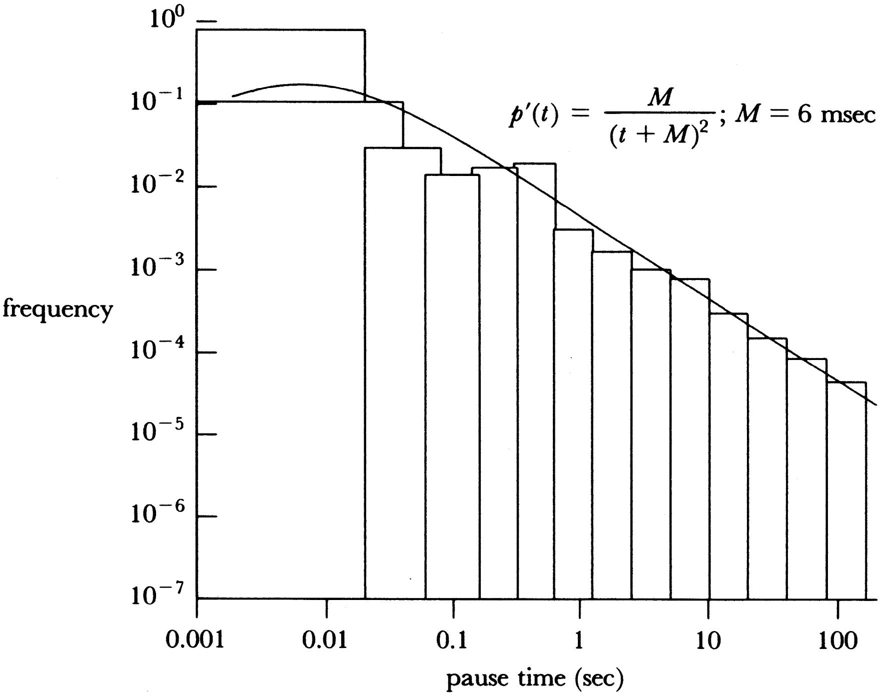


FIGURE 5.4. Measured pause-time distribution with distribution derived from $p'(t)$ superimposed

The pause-time distribution will be much more dependent on the underlying operating system than the run-time distribution. A distributed system like ***Amoeba*** may yield a probability density function with different characteristics altogether.

The primary cause for the "fat tail" distribution measured in UNIX is the usage of *getty* processes, pausing on a read from a terminal to log a new user in. At night, when hardly anyone is logged in, there are many such processes pausing for hours. A system that used a different method for logging in (*e.g.*, one process that handles all terminals) would undoubtedly show a pause-time distribution that was quite different.

5.3. A Model for a Distributed Scheduler

Now that a model for estimating processes' run times is available, it can be used in a scheduling algorithm. Assume there are N processes and K processors. To simplify the problem assume also that every process can be allocated to any processor, and that all processors are equally suitable for any process. A processor can execute one process at a time. No assumptions need to be made about where processes are when they are not allocated; they can be kept in primary memory somewhere or on a swapping device. The time to allocate and de-allocate a process depends on the method of storing de-allocated processes and is represented by the **switch time**, which may be process dependent.

When a processor is idle, the Scheduler can pick an unallocated process, and allocate it to the processor. When a processor is allocated, the Scheduler can de-allocate the process and allocate another (unallocated) process. In order to decide which process to allocate and which process to de-allocate we shall develop two priority functions, one for allocated processes and one for de-allocated processes. Using these priority functions we can decide which allocated process least deserves to be allocated and which unallocated process most deserves to be allocated, and we can decide if the least deserving allocated process is less deserving than the most deserving unallocated process.

The scheduler has three criteria to base its decisions on: to maximise throughput, minimise response times and ensure fairness, simultaneously, if possible. We shall briefly describe each criterion.

Maximise throughput

This criterion states that the amount of work done per unit of time must be as large as possible. Work can be measured in different ways, depending on what the Scheduler schedules. We shall use as a measure the number of instructions of the processes in the system executed per unit of time, where, for simplicity, we assume all processors are equally fast.

Minimise response times

The response time of a process is the time that elapses between the moment the process is created and the moment it delivers its results. In our model we do not consider the execution times of whole processes, but of individual runs. The response time then is the time that elapses between the moment a process becomes runnable (the start of a run) and the moment the process ceases to be runnable (through exit or pause). The response time of the system is the sum of the individual response times. Minimising response times will cause minimum wait times, both for human users at a terminal, for clients waiting for a service to be carried out, and for peripheral equipment.

Ensure fairness

> The throughput criterion and the response time criterion may not cause a runnable process never to be run at all. The fairness criterion states that processes or groups of processes must make equal progress. There are several ways to define the notion of fairness. One is, to give each process an equal share of the processor(s), another, to give each user an equal share, independent of the number of processes under his control.

These criteria are often contradictory: According to the throughput rule each processor must be kept as busy as possible, so processes with long expected run times should be run as much as possible, but according to the response time criterion, shortest jobs should be run first. Both rules may cause some processes never to be run at all, which violates the fairness rule. The scheduler cannot maximise throughput, minimise response times and ensure optimal fairness at the same time. It will have to enforce each of these rules to some degree, and, ideally, the weight of each rule is a parameter of the scheduler.

We shall phrase some scheduling rules, rules that increase throughput, decrease response times and ensure fairness. These rules will be based on the present allocation of processors to processes, and our predictions on the behaviour of processes in the near future. We shall not attempt an optimum schedule for any of the criteria, because it is not useful to compute—at great expense—an optimum allocation of processes, based on estimates of the involved processes' future behaviour.

Some terminology is needed before the scheduling rules can be described. Processes will be indicated by lower case letters. Predicted times are indicated by a capital T; times that can be computed (such as the time required to load a particular process in memory) by a lower case t. If a process, x, is runnable, the predicted remaining run time is T_x^r. If x is in a pause, the predicted remaining pause time is T_x^p. The time, required to load or unload process x is t_x^l.

5.3.1. Throughput Rule

Let us look at the throughput rule first. It simply states that the processors must be kept as busy as possible. This implies that, if a processor is idle and there is an unallocated runnable process, it should be assigned to that processor forthwith. It also implies that processes, once assigned to a processor should, if possible, run to completion, because process switching costs time. When an allocated process pauses, its processor is idle, so if the expected pause time exceeds the time required to exchange the pausing process with an unallocated runnable process, a switch should be made; when no unallocated runnable process is available, but there is an unallocated process with a shorter expected pause time than that of the allocated process, then a switch could also be useful. The variations in pause times are usually so large, however, that it is better to wait until a process becomes runnable. We have summarised the throughput rules in FIGURE 5.5; the rules are put in order of precedence, the top rule should be applied first, if applicable.

allocated process	unallocated process	switch if ...
No process	Runnable	Allocate the process with the longest expected run time.
	pausing	wait until a process becomes runnable.
pausing	runnable	switch if the expected pause time exceeds the switch time. Choose the process with the longest expected run time.
	pausing	wait until a process becomes runnable.
running	runnable or pausing	never switch a running process

FIGURE 5.5. Scheduling rules for maximising throughput, stated in order of precedence.

5.3.2. Response Time Rule

The next criterion we shall look at is response time. The response time of a run is the time between the moment a run starts—the moment the process becomes runnable—and the moment a run ends—the moment the process exits or starts a pause. The response time of a process consists of the time the process was allocated during a run, and the time it was not allocated. The total response time of the system is the sum of the response times of the individual processes.

Suppose there are two processes and one processor. Process one needs 100 seconds of CPU time to complete its run, while process two needs only 10 seconds. Switching processes costs one second. If process one is run first and process two after process one, the response time of process one will be 100 seconds, that of process two will be 100 + 1 + 10 = 111 seconds (the time waiting for process one, plus the time to switch processes around plus the time to run process two). The total response time is 211 seconds. It is easily verified that running process two first results in a total response time of 121 seconds. In order to obtain the best response time, the shortest process should be run first. It is not difficult to see that in a situation with many processors and processes the response time is also minimised by executing processes with shortest runs first.[10] When a processor is idle, and there is a choice of several unallocated runnable processes, the one with the shortest expected run time must be chosen.

When process x is allocated, with expected run time T_x, and process y is waiting to be allocated, with expected run time T_y, the expected response time if no process switch is made is

$$T_{resp} = 2T_x + T_y + t_l,$$

where t_l is the time it takes to unload process x and load process y. If the processes are switched, however, the response time will be

$$T'_{resp} = 2T^r_y + T^r_x + 2.t_l$$

The minimum response time rule thus states that a process switch should be made if $T_{resp} > T'_{resp}$, or

$$T^r_x - T^r_y > t_l$$

When the expected run times are accurate estimates, this rule indeed minimises response time. Consider, however, what happens when the remaining run time of a process is predicted using the amount of CPU time already consumed, according to Equation (5.2). A process, y, which ran for t^r_y CPU seconds has an expected run time of $\Upsilon(t^r_y) = 2.t^r_y + M$, hence an expected remaining run time of $T^r_y = t^r_y + M$. Assume process x is allocated and has had the CPU for $t^r_x = 20$ seconds, and process y is not allocated, and had the CPU for $t^r_y = 18$ seconds. Assume also that $t_l = 1$ second. Observe what happens if the minimum response time rule is applied:

$$T^r_x > T^r_y + t_l$$

$$t^r_x + M > t^r_y + M + t_l$$

$$20 > 18 + 1$$

This is true, so processes x and y are switched. Process y now needs about 18 seconds of CPU time to end its run, but as it executes, the expected remaining run time increases. After little more than 3 seconds of CPU time, T^r_y will be large enough to cause a new process switch, according to the response time rule stated above. Clearly, the response time rule must be changed to prevent processes being switched back and forth every $2.t_l$ seconds. This not only causes much unnecessary overhead and reduces throughput, it actually increases response time rather than reducing it.

Two things can be done to prevent this thrashing: one is to simply forbid processes to be switched back before the newly allocated process has had its remaining expected run time, the other is to change the response time rule in such a way that a switch will not be made unless the swapped-in process gets its remaining expected run time of CPU time, even though the expected run time changes. Note the difference between the two methods: the former method causes an immediate switch and prevents premature switching afterwards, the latter postpones switching, so that no special rules are needed afterwards.

We choose the latter method for two reasons. First, the former method requires special rules for processes that have just been allocated or have just been de-allocated. This means the scheduling history must be retained. Second, it requires the scheduler to explicitly ignore newer—hence, better—predictions of processes' run times after having made the switch. We have therefore chosen the latter method by formulating scheduling rules that will only make a switch if it is guaranteed that the involved processes will not be switched back before the newly allocated process has at least completed its expected run time. In order to formulate this rule, we view a process' expected remaining run time as a function of CPU time consumed, denoted by $\Upsilon'(t_y^r)$.

We now have two rules: the old minimum response time rule that states

$$T_x^r > T_y^r + t_l$$

and a new rule that states: assuming a switch is made, the above rule may not become applicable during a time of T_y^r, or

$$\Upsilon'(t) > T_x^r + t_l, \quad \textit{remains false} \text{ for } \textit{all } t_y^r \leqslant t \leqslant t_y^r + T_y^r$$

If the computation of the expected remaining run time, according to $\Upsilon'(t_y^r) = t_y^r + M$ is used, $\Upsilon'(t)$ is an increasing function of t, so the second rule becomes

$$\Upsilon'(t_y^r + \Upsilon'(t_y^r)) \leqslant t_x^r + M + t_l,$$

or

$$t_x^r \geqslant t_y^r + \Upsilon'(t_y^r) + M - t_l$$

If the original minimum response time rule is also written in terms of CPU time consumed, we get

$$t_x^r > t_y^r + t_l,$$

so we can combine the two rules to one new rule (substituting T_y^r for $\Upsilon'(t_y^r)$ once more):

$$t_x^r > t_y^r + T_y^r + M + t_l$$

We can now summarise the scheduling rules for minimising response time in a manner similar to FIGURE 5.5. The result is shown in FIGURE 5.6.

allocated process	**unallocated process**	**switch if ...**
No process	Runnable	Allocate the process with the shortest expected run time.
	pausing	wait until a process becomes runnable.
pausing	runnable	switch if $T_x^p > 2.t_l$.
	pausing	wait until a process becomes runnable.
running	runnable	switch if $T_x^r > T_y^r(t) + t_l$, for all $t_y^r < t < t_y^r + T_y^r$
	pausing	don't switch

FIGURE 5.6. Scheduling rules for minimising response time, stated in order of precedence.

5.3.3. Fairness Rule

The last criterion for scheduling is fairness. Each process, when runnable, has a right for an equal share of processing. This means that, ideally, at the end of a run, the ratio of the allocated time and the unallocated time should be the same for all processes. With N runnable processes and K processors, this ratio of allocated time to unallocated time should be K to $N-K$. Fairness can be achieved, for instance, by allocating processes for K units of time, after they have been de-allocated for $N-K$ units of time. By choosing a large unit of time, the amount of switching is small, but short processes, especially, are not treated fairly, because they do not use up their whole time slice. By choosing a short unit of time, however, process switching will take up all of the processor time. Ideally, both short and long processes get a fair share of the processors in as few time slices as possible.

In order to reduce the number of process switches and ensure fair treatment of both short and long runs, a scheduling rule is phrased that de-allocates a process when it has had its fair share of the CPU, and allocates a process when it will get its fair share if it executes for its expected run time. This will give processes with long expected run times long time slices at the price of having to spend a long time de-allocated, while processes with short expected run times will spend a short time being de-allocated, at the price of getting short time slices. Every process will still get a fair share of the CPU, however.

An allocated running process has received a fair share of the CPU if its *'degree of fairness,'*

$$D_{fair,alloc} = \frac{N-K}{K} \cdot \frac{t_x^r}{t_x^d} = 1 \qquad (K < N)\,,$$

where t_x^d is the time process x spent runnable, but de-allocated, waiting to be allocated. If this ratio is greater than one, the process has had more than its fair share; if it is less, it has had less. A (de-allocated) runnable process will have had its fair share at the end of its expected run time if its degree of fairness

$$D_{fair,dealloc} = \frac{N-K}{K} \cdot \frac{t_y^r + T_y}{t_y^d} = 1$$

If this ratio is less than one, the process will get its fair share after more than its expected time. If it is greater it will get it sooner. In order to give processes time slices of size expected remaining run time, we shall use $D_{fair,alloc}$ for allocated processes, and $D_{fair,dealloc}$ for de-allocated processes. Process x (allocated) and process y (de-allocated) should be switched if

$$\frac{t_x^r}{t_x^d} > \frac{t_y^r + T_y}{t_y^d}$$

Of the allocated processes, the first to be de-allocated is the one with the largest $D_{fair,alloc}$; of the de-allocated processes, the first to be allocated is the one with the smallest $D_{fair,dealloc}$.

There are no fairness rules for pausing processes or idle processors, but if a process has to be allocated, the fairness rule states that the process with the smallest degree of fairness should be chosen. We have summarised the fairness rules in FIGURE 5.7.

allocated process	**unallocated process**	**switch if ...**
No process	Runnable	Allocate the process with the smallest $D_{fair,dealloc}$.
pausing	runnable	fairness rule does not state that a process must be allocated, but if one is allocated, the one with the smallest $D_{fair,dealloc}$ should be chosen.
	pausing	fairness rule is not applicable in this case.
running	runnable	switch if $D_{fair,alloc} > D_{fair,dealloc}$.
	pausing	fairness rule is not applicable in this case.

FIGURE 5.7. Scheduling rules for ensuring fairness.

5.3.4. Combination of Rules

The scheduler must somehow combine the rules for the three criteria of throughput, response time, and fairness. Some of the rules are plainly contradictory: the throughput rule states: 'allocate the process with the longest expected run time,' while the response time rule says exactly the opposite. Still, the scheduler must use rules that give reasonably good throughput, reasonably good response times, and reasonable fairness.

Best throughput is obtained by making as few process switches as possible of runnable processes, and de-allocating pausing processes in favour of runnable ones. The rule never to de-allocate a running process will often contradict the fairness rules and response time rules, but de-allocating pausing processes in favour of runnable ones does not hurt fairness and can be good for obtaining better response times. To some degree we have already tried to obtain good throughput in the response time rules and fairness rules, by always trying to allocate processes for the duration of their expected run time. Thus the number of times a process is switched during a run can be kept low.

Most difficult to integrate are the response time rule and the fairness rule for switching runnable processes. A mixture of the two rules must be used, parameterised, so more stress can be given to fairness, or more to good response times, as the situation may demand. In general, for short processes, response time is usually more important than fairness, while for very long jobs, with runs of minutes or hours, fairness will be more important.

When the response time rule and the fairness rule for processes with expected run times according to EQUATION (5.1) are compared, a similarity between the two immediately catches the eye. The response time rule for switching an allocated running process, x, and a de-allocated runnable process, y, is:

$$t_x^r > t_y^r + T_y^r + M + t_l$$

and the fairness rule for such processes is:

$$\frac{t_x^r}{t_x^d} > \frac{t_y^r + T_y^r}{t_y^d}$$

The two formulas can be combined in several ways. One method to combine the two is to rewrite the fairness rule

$$t_x^r > \frac{t_x^d}{t_y^d} \cdot \left[t_y^r + T_y^r\right]$$

The rules can then be combined as follows:

$$t_x^r > \alpha \,.\, \frac{t_x^d}{t_y^d} \cdot \left[t_y^r + T_y^r\right] + (1-\alpha) \,.\, \left[t_y^r + T_y^r + M + t_l\right]$$

If $M + t_l$ is ignored, this rule can be further simplified to

$$t_x^r > \left[t_y^r + T_y^r\right] \cdot \left[\alpha \,.\, \frac{t_x^d}{t_y^d} + 1 - \alpha\right]$$

In this rule α is a weighting factor between fairness and response time. Setting α to one reduces the rule to the fairness rule, while setting it to zero gives the minimum response time rule. Intermediate values give a mixture between fairness and minimum response time.

Simulations have shown that different values for α were needed to obtain good scheduling for short processes and for long processes. In the competition between processes with very short run times and processes with average run times, short processes need to be more favoured by the response time rule than is the case in the competition between processes with average run times and processes with very long run times.

Another drawback of this weighting method is that it is not possible to write the rule in the form

$$P_\alpha(t_x^r, t_x^d) > P_\alpha(t_y^r, t_y^d)$$

where P is a priority function, parameterised with α. This is particularly important in a distributed environment, because it makes it possible that two schedulers can exchange lists of priorities for their processes. It also gives a sort of absolute measure for the priority of a process, independent of other processes.

A method for mixing the response time rule and the fairness rule that works much better is to use the following rule:

$$\frac{t_x^r}{t_x^d + D} > \frac{t_y^r + T_y^r}{t_y^d + D}$$

In this rule, D regulates the relative weights of fairness and response time. Setting D to zero reduces it to the fairness rule, setting it to a very large number, compared to the time a process is de-allocated during a run, reduces the rule to the response time rule. For short processes, spending short times de-allocated, D has more weight than for long processes, spending more de-allocated time, hence short processes are treated more according to the response time rule than long ones.

The important advantage of parameterisation with D, is that it does allow definition of a priority function, so we can not only decide which processes to switch, but also which allocated process most needs to be de-allocated, and which de-allocated process most needs to be allocated.

Another advantage of the second weighting method is that it allows an extra parameter for putting extra stress on throughput. This parameter, L, represents the time required to switch processes. If the switch time is short, L can be small, if it is long, L must be larger. The switch rule for a running and a runnable process is now:

$$\frac{t_x^r}{t_x^d + D} > \frac{t_y^r + T_y^r}{t_y^d + D} + L$$

In fact, L replaces the $M + t_l$ of the minimum response time rule.

The priority function for a running process can be defined by:

$$P_{L,D}(t_r,t_d) = \frac{t_r}{t_d + D}$$

and for a de-allocated runnable process:

$$P_{L,D}(t_r,t_d) = \frac{t_r + T_r(t_r)}{t_d + D} + L$$

The scheduling rules for combined fairness, response time and throughput can now be summarised as shown in FIGURE 5.8.

The ultimate test, of course, of different ways of combining fairness, response time and throughput is to look at their performance. We have run simulations, using different ways of weighting response time and fairness, varying the parameterisation, and also varying the number of processors and processes. We measured the response time of processes as a function of their CPU time. In a totally fair schedule, the ratio of the two is the same for all processes, because the response time of a process is the sum of its CPU time and its suspended time. The best response times are obtained when the response time is equal to the CPU time; in other words, when the ratio of response time and CPU time is equal to one. Measuring response time as a function of CPU time can therefore be used both to obtain an indication of response time and of fairness.

allocated process	unallocated process	switch if ...
No process	Runnable	Allocate the unallocated process with the smallest $P_{D,L}(t_r,t_d)$.
pausing	runnable	if the expected pause time is greater than the switch time, allocate the process with the smallest $P_{D,L}$; otherwise, use the switch rule for running processes.
	pausing	don't switch, wait until a process becomes runnable.
running	runnable	switch if $P_{D,L}(t^r_x,t^d_x) > P_{D,L}(t^r_y,t^d_y)$ for running process x and some unallocated runnable process y.
	pausing	don't switch.

FIGURE 5.8. Combined scheduling rules for ensuring fairness, minimising response time, and maximising throughput.

Simulation results are discussed in § 5.5.

5.4. Macro Runs and Pauses

In conventional time-sharing systems, when a process is blocked on a system call it can be swapped out to disk. For many system calls, the swapping time is much longer than the time required to execute the system call. It is therefore unwise to swap every process the moment it blocks, but only those that are likely to remain blocked for a time long enough to merit swapping.

Most time-sharing systems do this. This is because in most traditional time-sharing systems, the operating system can make a reasonable estimate of the time a process will remain blocked on a system call. In an open system like ***Amoeba***, this is not possible: processes make transactions with services outside the operating system; there is no telling in advance how long processes will block.

In a distributed system, where processes may be swapped over the network, swapping is an expensive operation. Therefore it is important to know something about the *expected pause time* of a process before swapping it. It is only worth while swapping processes with a pause time that is longer than the swap time. The problem is to find out in advance which pauses will last long enough to make swapping necessary.

In § 5.2, the pause-time distribution of the UNIX operating system was shown. In this section we shall use a different model for predicting pause times for two reasons: We believe the UNIX distribution has an exceptionally fat tail, due to the usage of *getty* processes waiting all night for some one to log in. The other reason is that the analysis further on in this section becomes simpler if we assume run times and pause times have the same distribution function (albeit with different parameters).

In this section we shall use a probability density function based on

$$p(t) = \frac{2M_p^2}{(t + M_p)^3}$$

that is, the same distribution function that was used for predicting run times. The mean expected pause time is M_p, and if the elapsed pause time is t_p, then the expected remaining pause time is $T_p = M_p + t_p$, from EQUATION (5.2).

Assume now, that T_m is the minimum expected pause time that makes swapping worth while. Then the scheduler need only consider processes with pause times longer than T_m; that is, processes that have already paused for a time $T_m - M_p$, or longer. Pauses of shorter duration are not considered by the scheduler.

A **macro run** is defined as an alternating sequence of runs and pauses, each pause shorter than $T_m - M_p$, ending with the first $T_m - M_p$ seconds of a pause that lasts longer than $T_m - M_p$. The rest of that last pause is defined as a **macro pause**. In contrast, the runs and pauses defined previously will be referred to as *micro* runs and pauses. The idea is illustrated in FIGURE 5.9.

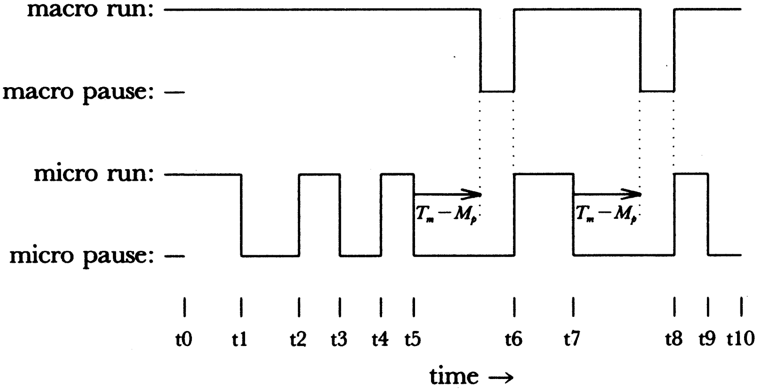


FIGURE 5.9. The relation between micro runs and pauses and macro runs and pauses. The pauses between $t5$ and $t6$, and between $t7$ and $t8$ are longer than $T_m - M_p$.

The distinction between micro runs and macro runs introduces a hierarchical notion of 'runnable': There can be schedulers at different levels of the system, using different notions of runs and pauses. One possibility, for instance, is to use a 'micro scheduler' to schedule processes within a processor, interacting with a 'macro scheduler' to schedule the processes of a collection of processors. When a process pauses for a short time, only the micro scheduler will know about it, and swap it for another *local* process. When the pause lasts longer, the macro scheduler is informed, which may swap the process over the network. The parameter T_m determines at which point processes come under the influence of the macro scheduler.

5.4.1. Analysis*

Given the run-time and pause-time distributions of EQUATION (5.1), let us try to analyse the probability distribution functions for macro runs and macro pauses:

Let $\underline{x}_i$, $i = 0, 1, 2, \cdots$, be random variables with probability density function

$$f(t) = \frac{2M_r^2}{(t + M_r)^3} \qquad (t \geqslant 0);$$

Let $\underline{y}_i$, $i = 0, 1, 2, \cdots$, be random variables with probability density function

$$g(t) = \frac{2M_p^2}{(t + M_p)^3} \qquad (t \geqslant 0);$$

The random variable $\underline{n}$ has the value n, $n = 0, 1, 2, \cdots$ with probability

$$Pr\{\underline{y}_i < T_m \ (i = 1, 2, \cdots, n) \quad \text{and} \quad \underline{y}_{n+1} \geqslant T_m\} ,$$

that is, with

$$\beta \equiv Pr\{\underline{y}_i < T_m\} = \int_0^{T_m} g(t)\,dt = 1 - \left[\frac{M_p}{M_p + T_m}\right]^2 ,$$

one has

$$Pr\{\underline{n} = n\} = (1 - \beta)\beta^n \quad , n = 0, 1, \cdots .$$

Finally, let

$$\underline{p} = \underline{x}_0 + \sum_{i=1}^{n} (\underline{y}_i + \underline{x}_i) + T_m ,$$

as illustrated in FIGURE 5.10. Let $\underline{p}' = \underline{p} - T_m$.

Then, with

$$P(t) = Pr\{\underline{p} < t\} \quad (p(t) = \frac{\mathrm{d}}{\mathrm{d}t} P(t))$$

[*] This analysis is due to E. van Doorn

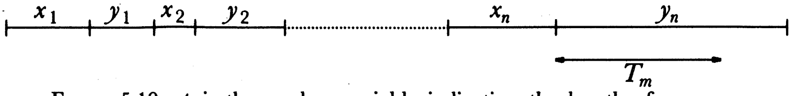


FIGURE 5.10. p is the random variable indicating the length of a macro run, composed of micro runs $\underline{x}_i$ and micro pauses y_i, where each y_i, $(i = 1, \cdots, n)$ is shorter than $\underline{T}_m$, but $\underline{y}_{n+1}$ is longer than T_m.

one has

$$P'(t) = (1-\beta)\sum_{i=0}^{\infty}\beta^n Pr\{\underline{x}_0 + \underline{y}_1 + \underline{x}_1 + \cdots + \underline{y}_n + \underline{x}_n < t \mid y_i < \underline{T}_m\}$$

Letting

$$f^*(s) = \int_0^{\infty} e^{-st} f(t)\,\mathrm{d}t$$

be the Laplace transform of $f(t)$,

$$\tilde{g}^*(s) = \frac{1}{\beta}\int_0^{T_m} e^{-st} g(t)\,\mathrm{d}t$$

$$p^*(s) = \int_0^{\infty} e^{-st} p(t)\,\mathrm{d}t \quad \left(= \int_0^{\infty} e^{-st}\,\mathrm{d}P(t)\right)$$

it follows that

$$p^*(s) = (1-\beta)\sum_{n=0}^{\infty}\beta^n\{f^*(s)\}^{n+1}\{\tilde{g}^*(s)\}^n$$
$$= \frac{(1-\beta)f^*(s)}{1-\beta f^*(s)\tilde{g}^*(s)}\,.$$

It is readily verified that

$$f^*(s) = 2e^{M_r s}E_3(M_r s)$$

and

$$\beta\tilde{g}^*(s) = 2e^{M_p s}\left\{E_3(M_p s) - \left[\frac{M_p}{M_p + T_m}\right]^2 E_3\left[(M_p + T_m)s\right]\right\}$$

where

$$E_n(z) = \int_1^{\infty}\frac{e^{-zt}}{t^n}\mathrm{d}t \qquad \text{(exponential integral)}$$

Since

$$E_3(z) = \frac{1}{2} - z - \frac{1}{2} z^2 \ln z + O(z^{2)} \quad (z \to 0)$$

([1], 5.1.12), it follows after some algebra that, for $(s \to 0)$,

$$p^*(s) = 1 - \left\{ M_r \left[1 + \frac{T_m}{M_p} \right]^2 + \frac{b^2}{M_p} \right\} s - M_r^2 \left[1 + \frac{T_m}{M_p} \right]^2 s^2 \ln s + O(s^2)$$

Consequently, the expected duration of a macro run is:

$$M_m = \int_0^\infty t\, p(t) \mathrm{d}t = - \left\{ \frac{\mathrm{d}}{\mathrm{d}s} p^*(s) \right\}_{s=0} = M_r \left[1 + \frac{T_m}{M_p} \right]^2 + \frac{T_m}{M_p} + T_m.$$

One also has

$$p(t) \sim \frac{2M_r \left(1 + \frac{T_m}{M_p}\right)^2}{t^3} \quad (t \to \infty) \qquad (5.3)$$

Apparently, for large t, the probability density function for macro runs is very similar to that for micro runs. A mathematical analysis for smaller t is much harder; instead we have analysed the behaviour by simulation.

The probability density function for macro runs depends on M_r, M_p and the choice of T_m. In our simulations we have generated random sequences of runs and pauses with expected durations of 26.4 msec and 160 msec, respectively, in accordance with the measured values mentioned earlier. From these sequences the macro runs and macro pauses were determined using various values of T_m. The minimum duration of a macro run is, of course, T_m, which shows in the distribution graphs by an apparent shift to the right of the whole graph.

5.5. Simulation Results

A simulator was used to obtain some results that indicate the usability of the scheduling rules formulated in the previous section. The program could simulate the execution of a number of processes on a number of processors. Each process alternated runs and pauses, whose lengths were drawn randomly according to the distribution of EQUATION (5.1). We have already shown that this function accurately yields the run time distribution of actual processes. It does not reflect the pause time distribution very accurately, but when other distributions were tried in the simulator the results were not significantly different.

Runnable processes compete for the available processors; pausing processes do not compete for processors. The process switch time was assumed to be zero in the simulations. The simulator used two priority functions for runnable processes, one for processes allocated to a CPU (swapped in), one for processes not

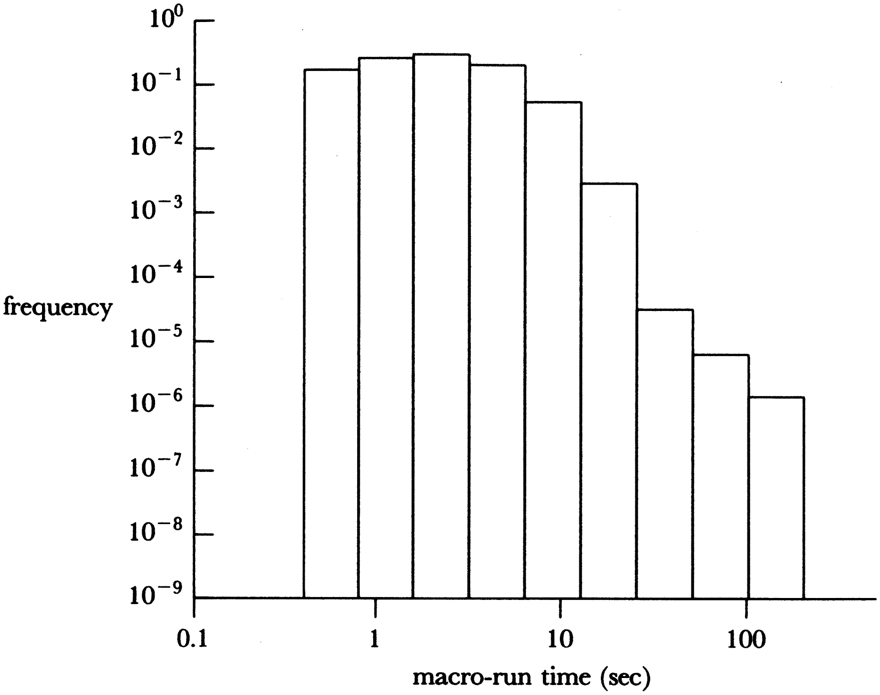


FIGURE 5.11. The distribution of macro runs and pauses as generated by simulation. (M_r = 26.4 msec, M_p = 160 msec, T_m = 500 msec.)

allocated to one of the CPUS (swapped out). For swapped-in processes the priority function was

$$P_{in} = \frac{t_r}{t_d + D}$$

while for swapped-out processes

$$P_{out} = \frac{t_r + T_{\exp}}{t_d + D} + L$$

was used. Note that the higher the value of the priority function, the more CPU time it has had, hence the less it is entitled to be or remain allocated. D and L are adjustable parameters of the priority function, D can be used to adjust the relative importance of minimising response time and ensuring fairness, L can be set to slightly delay switching processes to obtain better throughput. The simulator continuously compared the highest priority of the swapped-in processes to the lowest priority of the swapped-out processes, and, if the former was greater, switched the corresponding processes. Running processes that finished their run were, of course, immediately replaced by the swapped-out runnable process with the lowest priority.

The simulator was run with varying values of D and L to obtain some insight into their behaviour and to find optimal values. T_{exp} was computed according to the rule of EQUATION (5.1).

At the end of the simulation various statistics were printed. The performance of the scheduler can best be judged from two of these, shown on the following pages: One gives the ratio of the actual response time and the *fair* response time, that is, the response time each process would have had if all processes received exactly the same fraction of the CPU during a run. The other gives the average number of times the process was swapped out during a run. Both graphs are gives as a function of the run length of processes. The first one can be used to check the performance of the scheduler in terms of both fairness and response time. Optimal fairness is achieved if the function is constant; good response time is achieved if the function increases, that is, shorter processes get better response time. The second graph gives an indication of the overhead of the scheduler. The number of swaps should be kept as low as possible, since swapping is a relatively expensive operation, especially, if swapping must be done through the network.

Each figure shows scheduling results for one and for five processors under two different loads: A lightly loaded system: 5 processes per processor, where a process has pauses that last about six times as long as runs, and a heavily loaded system with 10 processes per processor. L and D were normalised by multiplying them with a normalisation factor L' and D', respectively:

$$L' = \frac{k}{n} \cdot \frac{M_p + T_m}{M_m}$$

$$M' = \frac{n}{k} \cdot \frac{M_m^2}{M_m + M_p + T_m}$$

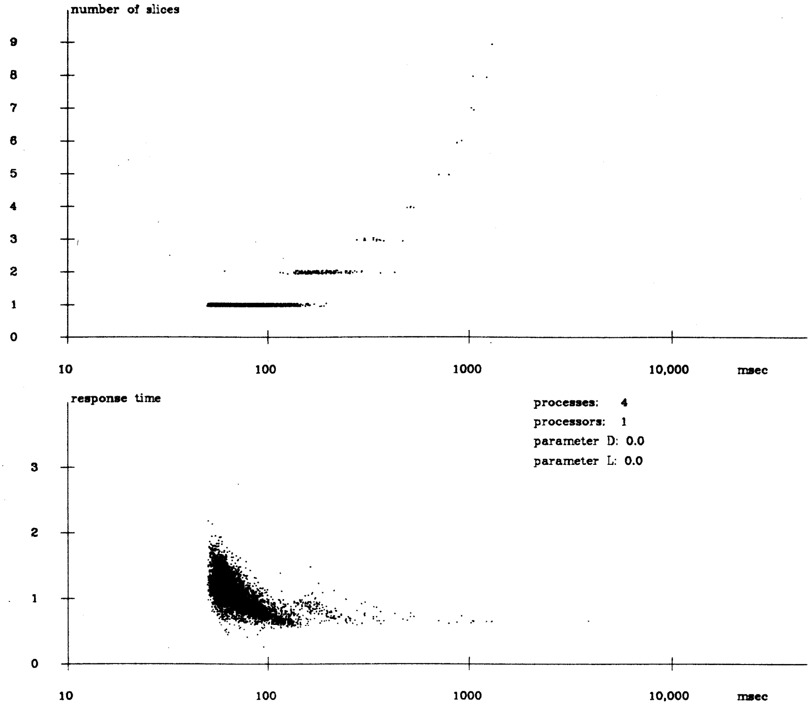
where n is the number of processors and k the number of processes.

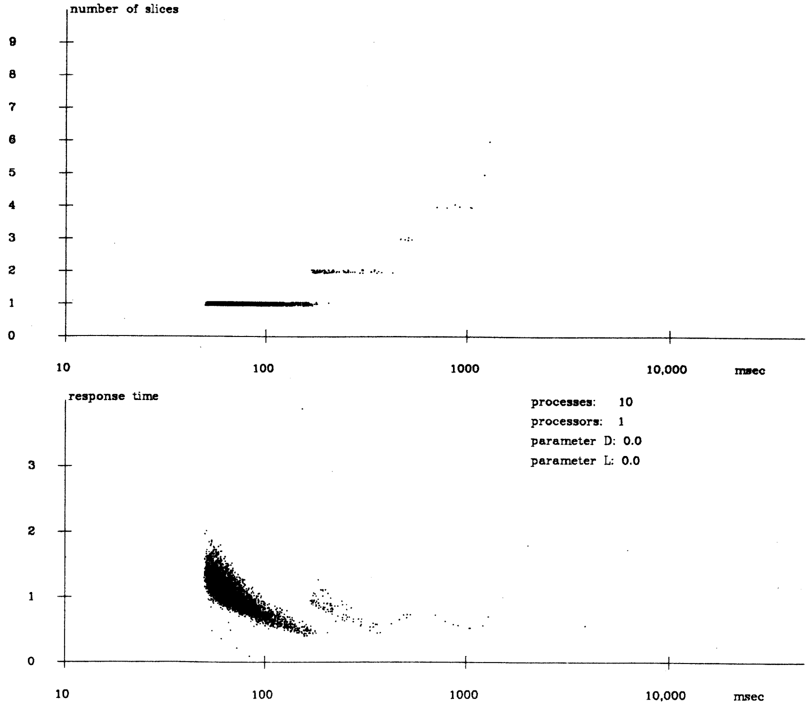
In the first figure, both L and D are set to 0, in the second, L is set to $3L'$, in the third, D is set to D' and in the last, L is set to $3L'$ and D to D'.

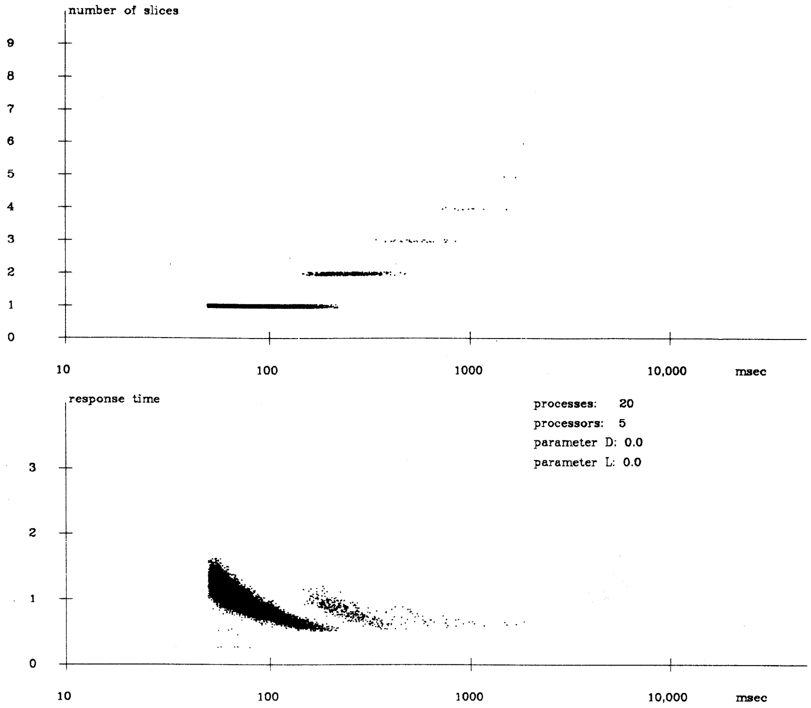
In all figures, the response time graph resembles a *sawtooth* wave. This is caused by swapping: processes that finish their run just before they are swapped have a better response time than processes that are swapped just before they finish their run. Within a single 'sawtooth wave,' shorter processes have relatively less good response times, due to the fact that they run for shorter times than estimated by the scheduling algorithm.

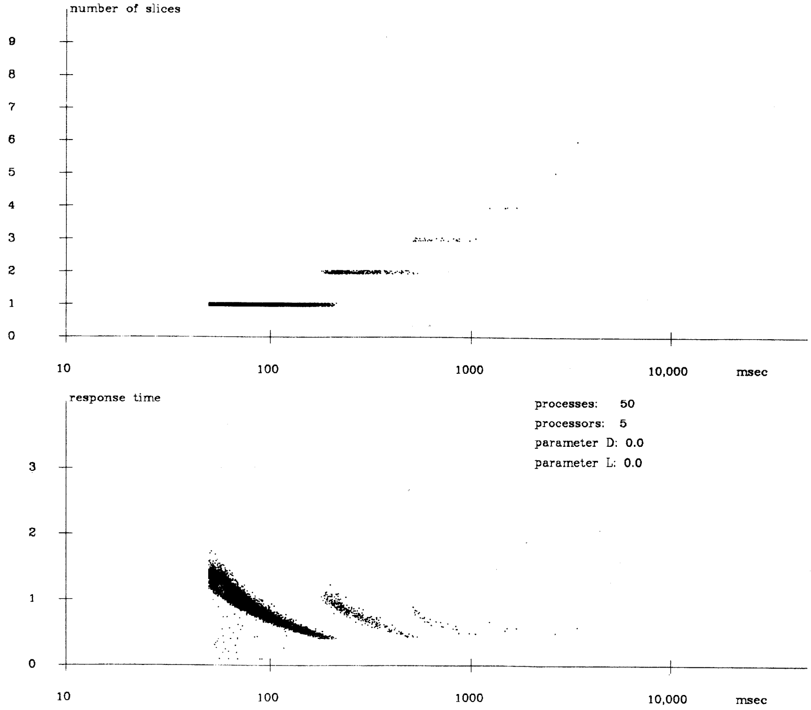
Setting D improves response time: shorter processes have relatively better response time than longer ones; each sawtooth wave is higher than its

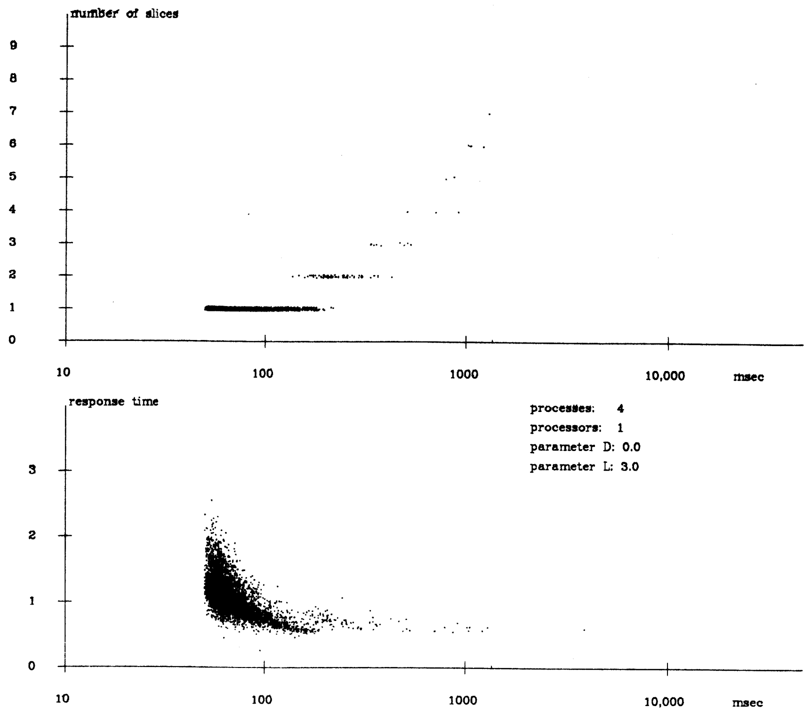
FIGURE 5.12—5.15 (following pages). Simulation results for scheduling various numbers of processes and processors using various scheduling parameters.

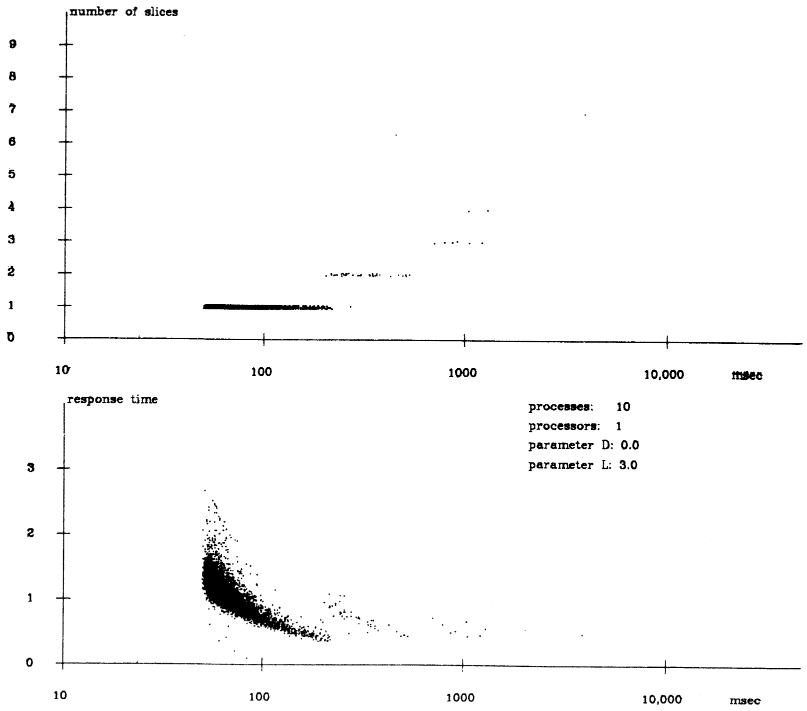

number of slices
9
8
7
6
5
4
3
2
1
0
10
100
1000
10,000
msec
response time
processes: 4
processors: 1
parameter D: 0.0
parameter L: 0.0
3
2
1
0
10
100
1000
10,000
msec

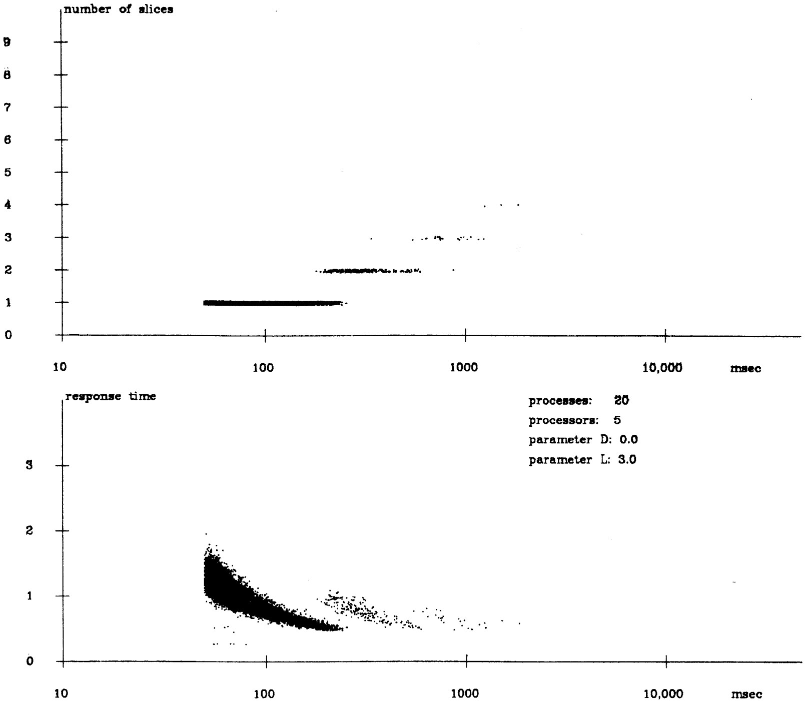

number of slices
9
8
7
6
5
4
3
2
1
0
10
100
1000
10,000
msec
response time
processes: 10
processors: 1
parameter D: 0.0
parameter L: 0.0
3
2
1
0
10
100
1000
10,000
msec

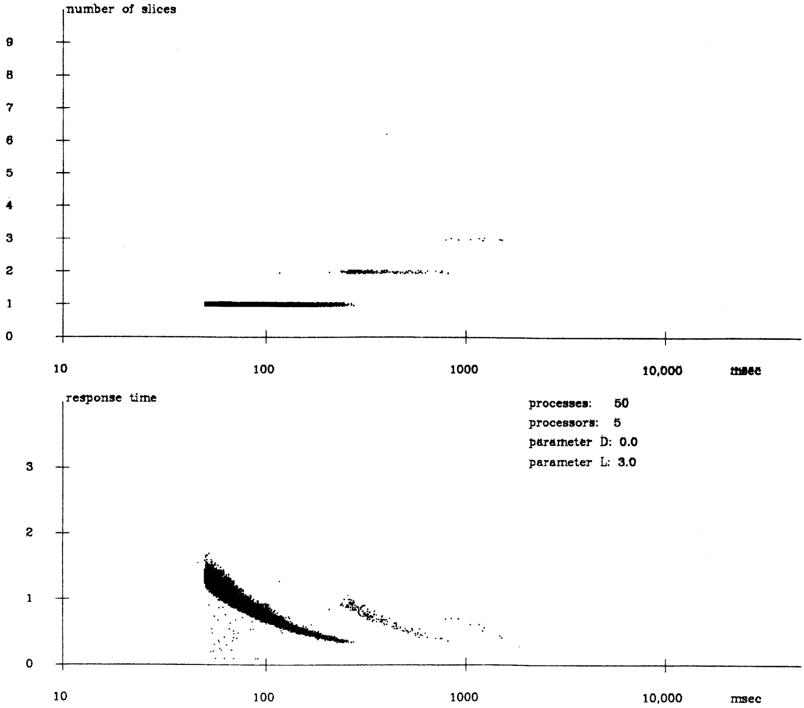

number of slices
0
1
2
3
4
5
6
7
8
9
10
100
1000
10,000
msec
response time
processes: 20
processors: 5
parameter D: 0.0
parameter L: 0.0
0
1
2
3
10
100
1000
10,000
msec

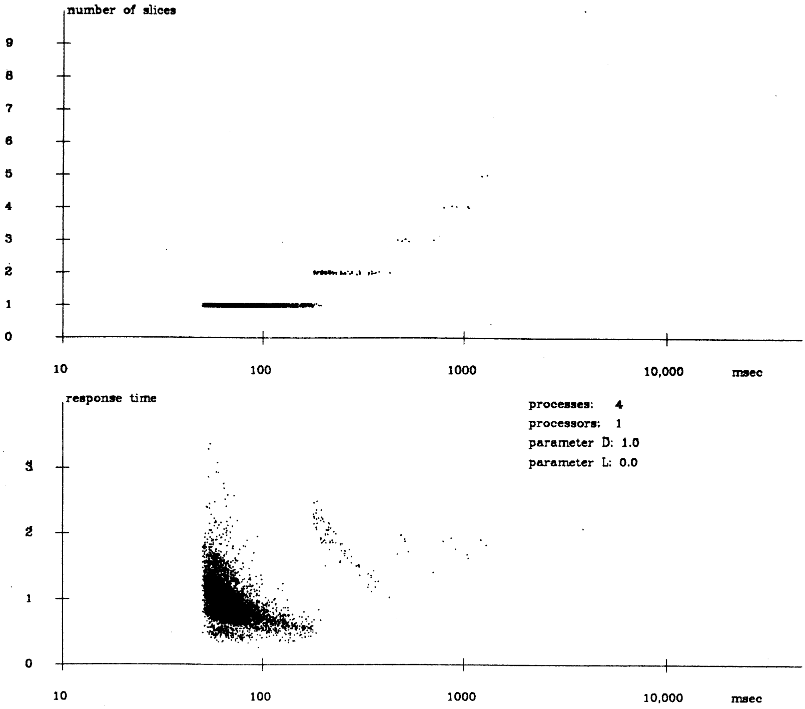

number of slices
0
1
2
3
4
5
6
7
8
9
10
100
1000
10,000
msec
response time
processes: 50
processors: 5
parameter D: 0.0
parameter L: 0.0
0
1
2
3
10
100
1000
10,000
msec

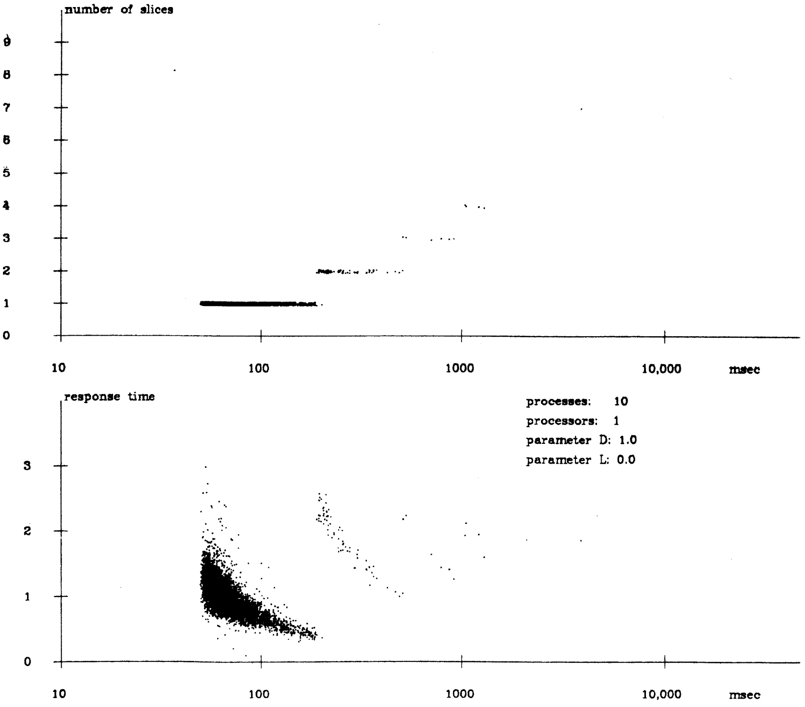

number of slices
9
8
7
6
5
4
3
2
1
0
10
100
1000
10,000
msec
response time
processes: 4
processors: 1
parameter D: 0.0
parameter L: 3.0
3
2
1
0
10
100
1000
10,000
msec

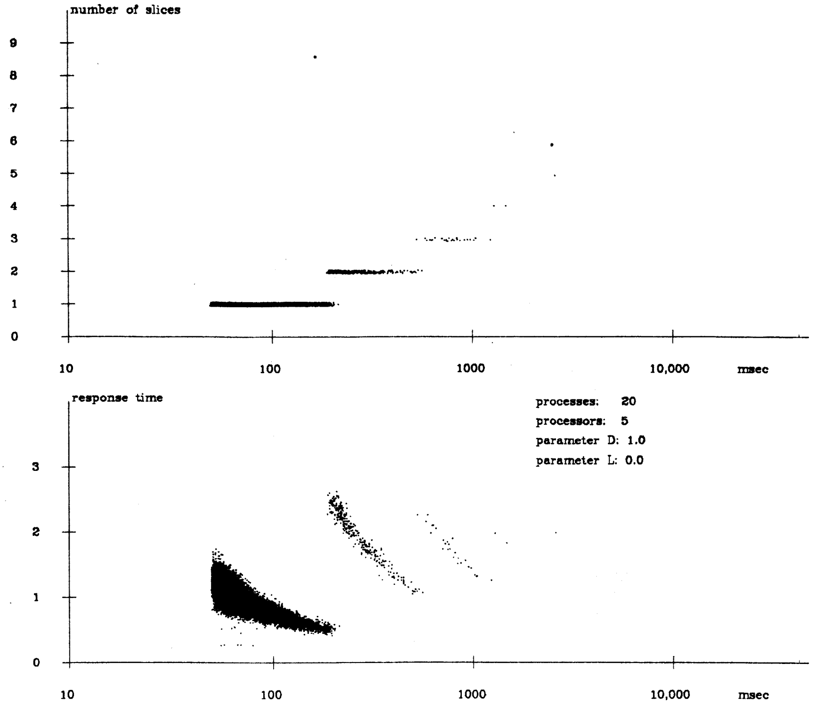

number of slices
9
8
7
6
5
4
3
2
1
0
10
100
1000
10,000
msec
response time
processes: 10
processors: 1
parameter D: 0.0
parameter L: 3.0
3
2
1
0
10
100
1000
10,000
msec

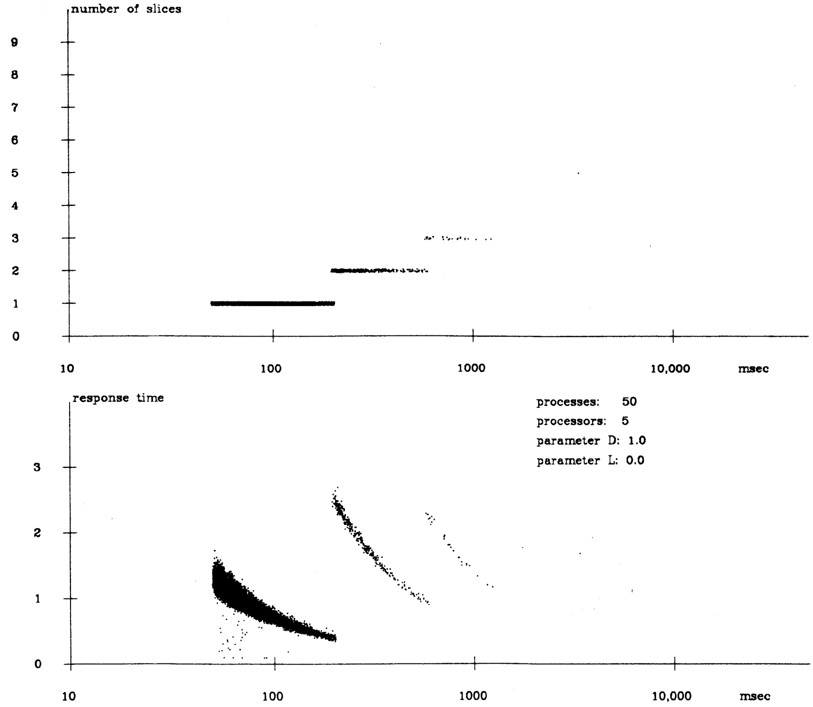

number of slices
9
8
7
6
5
4
3
2
1
0
10
100
1000
10,000
msec
response time
processes: 20
processors: 5
parameter D: 0.0
parameter L: 3.0
3
2
1
0
10
100
1000
10,000
msec

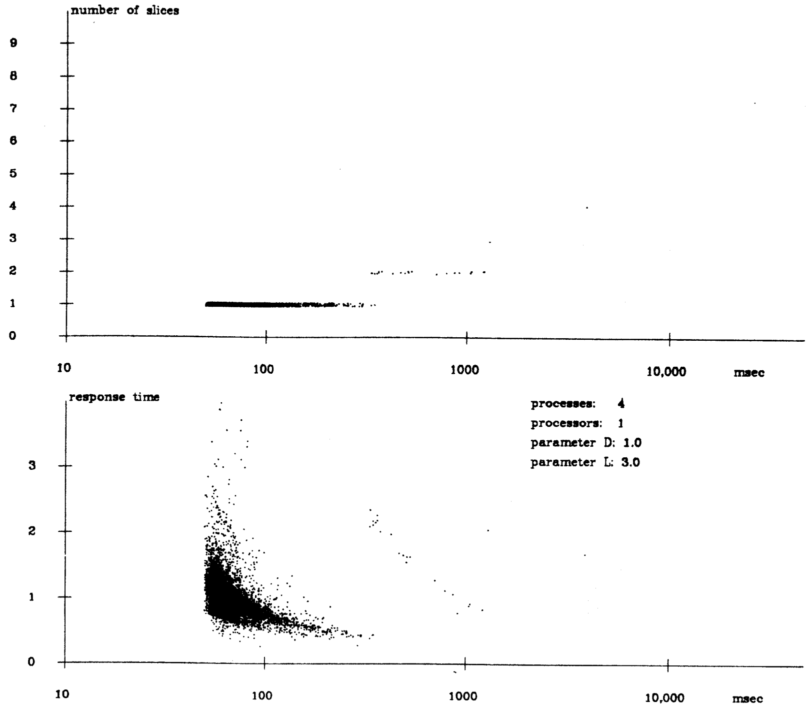

number of slices
9
8
7
6
5
4
3
2
1
0
10
100
1000
10,000
msec
response time
processes: 50
processors: 5
parameter D: 0.0
parameter L: 3.0
3
2
1
0
10
100
1000
10,000
msec

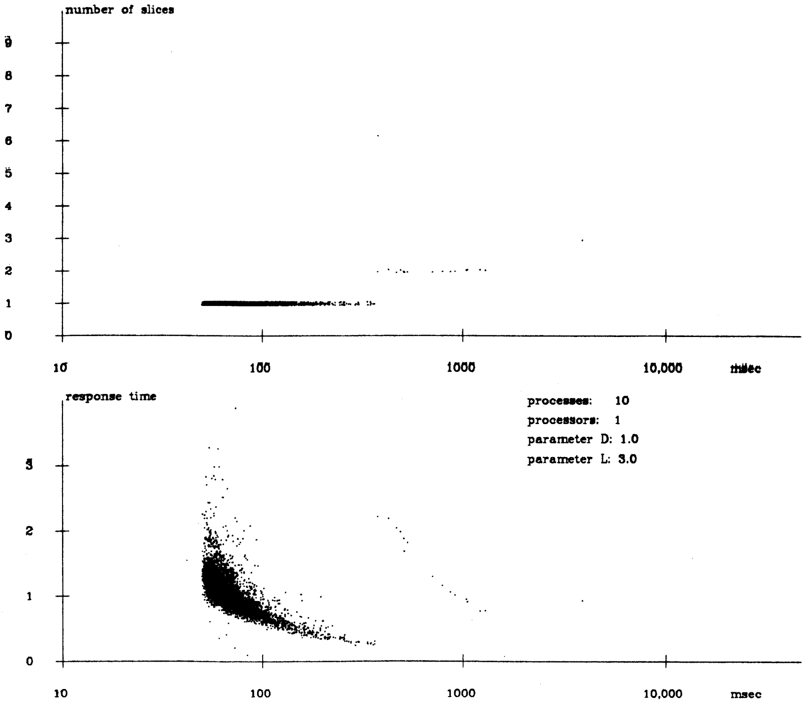

number of slices
9
8
7
6
5
4
3
2
1
0
10
100
1000
10,000
msec
response time
processes: 4
processors: 1
parameter D: 1.0
parameter L: 0.0
3
2
1
0
10
100
1000
10,000
msec

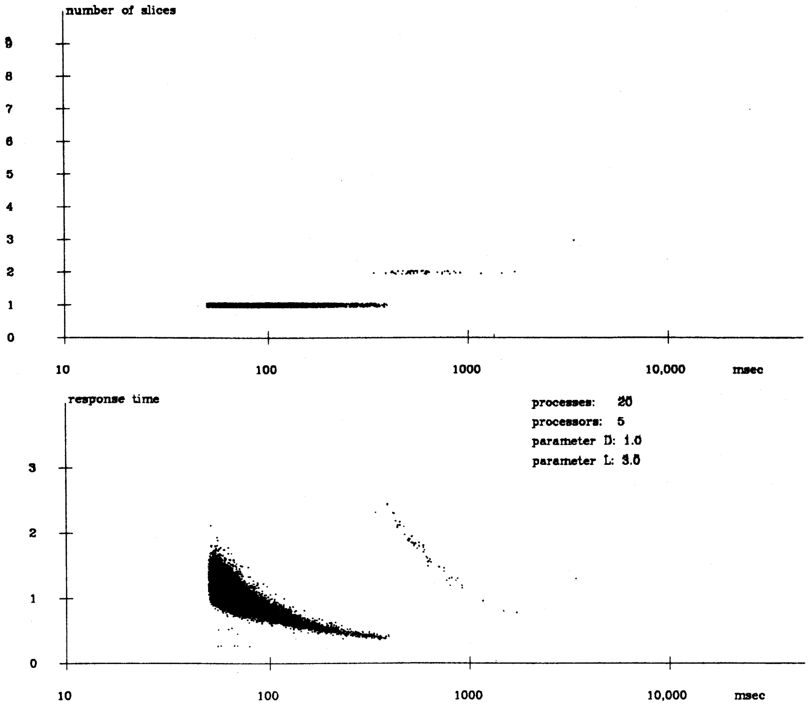

number of slices
9
8
7
6
5
4
3
2
1
0
10
100
1000
10,000
msec
response time
processes: 10
processors: 1
parameter D: 1.0
parameter L: 0.0
3
2
1
0
10
100
1000
10,000
msec

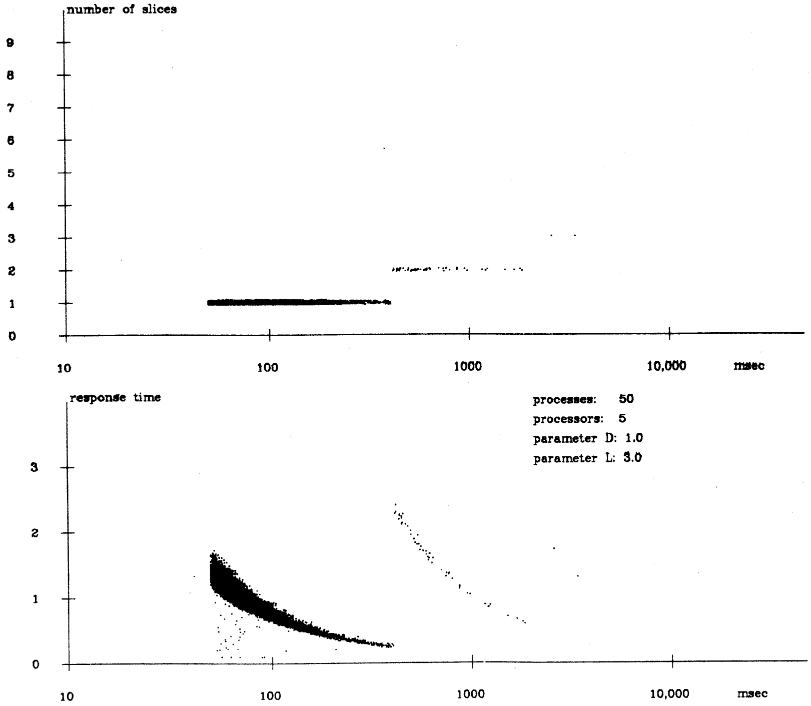

number of slices
9
8
7
6
5
4
3
2
1
0
10
100
1000
10,000
msec
response time
processes: 20
processors: 5
parameter D: 1.0
parameter L: 0.0
3
2
1
0
10
100
1000
10,000
msec


number of slices
9
8
7
6
5
4
3
2
1
0
10
100
1000
10,000
msec
response time
processes: 50
processors: 5
parameter D: 1.0
parameter L: 0.0
3
2
1
0
10
100
1000
10,000
msec


number of slices
0
1
2
3
4
5
6
7
8
9
10
100
1000
10,000
msec
response time
0
1
2
3
processes: 4
processors: 1
parameter D: 1.0
parameter L: 3.0
10
100
1000
10,000
msec


number of slices
0
1
2
3
4
5
6
7
8
9
10
100
1000
10,000
msec
response time
0
1
2
3
processes: 10
processors: 1
parameter D: 1.0
parameter L: 3.0
10
100
1000
10,000
msec


number of slices
9
8
7
6
5
4
3
2
1
0
10
100
1000
10,000
msec
response time
3
2
1
0
10
100
1000
10,000
msec
processes: 20
processors: 5
parameter D: 1.0
parameter L: 3.0


number of slices
9
8
7
6
5
4
3
2
1
0
10
100
1000
10,000
msec
response time
3
2
1
0
10
100
1000
10,000
msec
processes: 50
processors: 5
parameter D: 1.0
parameter L: 3.0

neighbours on the left, the response time curve increases. L influences the number of slices a process needs to complete its run. Making L larger has a deteriorating influence on the response time of very short runs, but increases the length of time a process remains allocated to a processor. Setting L increases throughput.

Research in the area of process scheduling is still being carried out in the Amoeba project. A Processor Pool will be built to test this scheduling algorithm, and perhaps others as well. The simulation results presented above are preliminary results: process run-time and pause-time distributions will have to be measured again when ***Amoeba*** is operational in a production-environment to see if our results are still valid. Another point of interest is to see how well a processor pool can be scheduled when no information is available on the length of pauses, information that is usually available—albeit not very accurately—in traditional operating systems.

5.6. Examples of Scheduling

5.6.1. Scheduling Multiprogrammed Processors

Many general-purpose operating systems manage many processes simultaneously on a single CPU. A runnable process is assigned to the CPU where it executes instructions until it ceases to be runnable, or the clock signals the end of its time slice. The decision which process to assign to the CPU can be made using the priority function derived in the previous section.

In many operating systems, however, there is an additional difficulty in scheduling, because of the limited amount of main memory that processors have. Not all processes can reside in main memory at the same time, so some processes must be swapped out to secondary memory.

One approach to allocating memory for processes is to use *demand paging*.[14] Demand paging can easily be combined with our scheduling algorithm: when a page fault occurs, the process stops being runnable, and a pause starts, the length of which can be predicted quite accurately. Process switching will cost far less time than the time it takes to fetch a page from disk, so the runnable process with the best priority is chosen to be run next. When no process is runnable, the scheduler must wait until a process does become runnable.

In other operating systems, usually running on hardware that does not support demand paging, other methods of allocating processes to main memory are needed. Often, in these systems, processes are swapped in and out as a whole. In these systems, of which many versions of UNIX are examples, a process resides wholly in memory, wholly on the swapping device, or it is in the process of being transferred from one to the other. In time sharing systems, such as these, two levels of scheduling are necessary: it has to be decided which process should be swapped in or out, and which of the processes in main memory should be allocated to the processor.

The **CPU scheduler** decides which of the processes in main memory to allocate to the CPU. The scheduling algorithm for this can be the one of the previous section. The **swap scheduler** must decide which processes to swap. If they all had equal size, the same algorithm would have been suitable also for deciding which processes to swap in or out. This is never the case, however. One swapped out process can make room for several processes to be swapped in, or several processes may have to be swapped out to swap one in. All other things being equal, it is best to swap the smallest process in, and the largest out. The swap scheduler must not only take priorities into account, but also the sizes of the processes to be swapped.

As far as the swap scheduler is concerned, primary memory is a scarce resource, so a measure of primary memory use is the amount of memory the process occupies, multiplied by the time it occupies that memory. Like the CPU scheduler, the swap scheduler is also concerned with obtaining maximum throughput, minimum response time, and ensuring fairness. Again, we shall examine them in turn.

For the size of a process, x, we shall write s_x. For the time a process spent in primary memory during a run we write $t_{in,x}$, and for the time swapped out $t_{out,x}$.

Maximum throughput can be obtained by ensuring that at all times at least one runnable process is in primary memory, so the CPU will not be idle. This can be achieved by trying to ensure that there is always at least one runnable process in primary memory besides the one that is currently executing, but this is only possible if enough memory is available. The swap scheduler must have as many processes in primary memory as possible, and never swap out the currently running process.

Minimising response time means that processes occupying little memory with short expected run times should have precedence over processes needing much memory with long expected run times. We shall use the same criterion for swapping processes as for CPU scheduling, replacing $T_{r,x}$ by $T_{in,x} . s_x$. The time required to swap processes in or out is proportional to the size of the process, so we replace t_l by $t_{swap} . s_x$ where t_{swap} is the time required to swap a process of unit size.

If we do the same substitution in the fairness formula as in the one for response time, we get the following rule for swapping two processes:

$$\frac{t_{in,x}}{t_{out,x} + U} . s_x > \frac{t_{in,y} + T_{in,y}}{t_{out,y} + U} . s_y$$

where $T_{in,y}$ is the estimated time process y must be in primary memory to complete its run. If there are on average k processes in primary memory, $T_{in,y} = k.T_{r,y}$.

The swapping rule as stated above has two aspects that need closer inspection. The first is that the rule considers primary memory as the only resource to be scheduled. This is not the case. It is the CPU that is the essential resource.

Scheduling of primary memory must be subordinate to scheduling of the CPU. The rule as stated above will allocate a process to primary memory twice as long as another of twice the size. Small processes will therefore have much more chance of being allocated to the CPU, since the CPU scheduler can only choose from the allocated processes. A weighting parameter, S should be added to the scheduling rule to reduce the importance of a process' size:

$$\frac{t_{in,x}}{t_{out,x} + U}.(s_x + S) > \frac{t_{in,y} + T_{in,y}}{t_{out,y} + U}.(s_y + S)$$

The second aspect also has to do with the unequal size of processes. The rule cannot be used to decide to swap one process out and another in, because the hole left by the swapped out process may not be big enough for the process to be swapped in, or it may be big enough for several. The rule should become: Find the process, y, most in need of swapping in; that is, the process with the smallest

$$P_{out,y} = \frac{t_{in,y} + T_{in,y}}{t_{out,y} + U}.(s_y + S)$$

If there is room in primary memory, swap the process in. If not, find the processes in primary memory with

$$P_{in,x} = \frac{t_{in,x}}{t_{out,x} + U}.(s_x + S)$$

greater than $P_{out,y}$. If these occupy more room than process y needs, swap as many of them out to make room for it, then swap it in. If they occupy less room, wait until there are enough processes with $P_{in,x} > P_{out,y}$ to make room for process y.

5.6.2. Scheduling Pool Processors

In § 4.3, the mechanisms used by the Process Servers to execute processes on Pool Processors were described. Using these mechanisms, the Process Service can implement the scheduling rules that we developed in § 5.3. Each Process Server will have a number of Pool Processors under its control, and receive requests to execute processes. If there is just one Process Server in the system, this method works well. Even if the Process Server crashes, the Boot Service of § 4.5 can bring Process Service back to life quickly enough not to cause great inconvenience to its clients.

With more than one Process Server we have the advantage of increased accessibility of Process Service, and greater capacity. But it presents an additional problem to be solved: distributing the work evenly over the Process Servers. To do this a measure of the workload on a Process Server is needed.

The workload on a Process Server and its Pool Processors consists of two parts, the workload on the Pool Processors, and the overhead for the Process Server itself. The Pool Processor workload depends on the number of Pool Processors available to do the work, the number of processes that run on these Pool Processors, the ratio of run time and pause time of the processes, the number of process switches necessary, and the time needed for process switching, which depends on the size of the processes. The Process Server overhead depends primarily on the number of process switches, which depends, among other things, on the number of processes.

To measure the Pool Processor workload we introduce the *weight* of a process, a measure for the load a process puts on a Pool Processor. The weight of a process, x, is

$$w_x = \frac{\sum t_{r,x}}{\sum t_{r,x} + \sum t_{p,x}}$$

where we sum over the runs and pauses in the process' history up to now. The weight of a totally compute bound process is thus one, while the weight of a forever-pausing process, which—assuming the process can be swapped out—places no load on any Pool Processor, is zero. Note that this measure depends on the characteristics of the process, and the events it pauses on, not on the way the process is scheduled.

The load on the Process Server is mainly determined by the number of process switches it must handle. A measure for this is the number of state changes (pause to run, and run to pause) per unit of time of the processes under its control. For the Process Server load caused by a process, x, we use

$$w'_x = \frac{N_{r,x}}{\sum t_{r,x} + \sum t_{p,x}},$$

where $N_{r,x}$ is the number of runs in its history.

The work load on a Process Server and its Pool Processors is a weighted sum of the work load on the Process Server and that on the Pool Processors of the processes under the Process Server's control:

$$W = \frac{\delta \,.\, \sum_x w_x + (1-\delta) \,.\, \sum_x w'_x}{n}$$

where n is the number of processes. By adjusting δ, the relative weights of Pool Processor load and Process Server load can be set.

The work load measure that has just been defined can be used to distribute processes and Pool Processors more evenly over the Process Servers. In some cases, it is simplest to allocate more Pool Processors to busy Process Servers, taking them away from less busy ones, in others it is more convenient to migrate processes from busy Process Servers to less busy ones. In multi-programmed machines, for instance, the number of (virtual) Pool Processors is usually fixed, and one special process functions as Process Server. In a situation like this, it is not possible to reallocate Pool Processors; load balancing must be achieved by reallocating processes to other Process Servers. A different situation arises when a Process Server controls a number of microcomputers—each functioning as a Pool Processor—over a network; in this case it may be more convenient to pass Pool Processors from Process Server to Process Server.

Two approaches exist to achieve load balancing. One method is to let the Process Servers exchange information periodically to compare work loads. If the difference is greater than some predetermined amount, processes or processors should be exchanged. Another method is to create a service especially for load balancing. Such a service would periodically obtain the work load of each of the Process Servers connected to it, and balance the load by ordering the Process Servers to pass processes and Pool Processors around. Potentially, a separate load balancing service has more global information than the individual Process Servers in the former method, but its centralised approach makes it more vulnerable to crashes. The former method is simpler and, if the load distribution does not change very rapidly, it will work just as well.

6

THE DISTRIBUTED FILE SERVER

File systems play an important role in allowing information to be widely accessible, since most information is in one way or another stored in files. Many different kinds of file systems for distributed systems exist, ranging from private file systems for each host to special purpose file servers for the whole network. Each kind of file system has its own characteristics concerning accessibility, complexity, protection of information against unauthorised access, speed and distributiveness.

The ideal distributed file system would be fast, files would always be near the hosts needing them, there would be protection, if necessary, to guard against access from unauthorised hosts or users, files could be shared among different hosts at the same time, and the system would be totally immune agains individual file server crashes or disk crashes. Unfortunately, such distributed file systems do not yet exist, and improving one aspect of a file system is nearly always detrimental to another. The consequence, for instance, of replicating files at several sites to improve their availability is that updating these files will become more costly, since all copies have to be updated, and if, additionally, the changes made by different users must be synchronised, such that the changes made by one user do not interfere with the data read by another, then the cost of file operations could be increased by several orders of magnitude.

This chapter goes into the design of the Amoeba File Service, one of the three file services for the Amoeba Distributed System. § 6.1 describes the considerations that led to the design of this file server and gives an overview of related work. The underlying Block Service is briefly discussed in § 6.2. A detailed description of the Amoeba File Service follows in § § 6.3.

6.1. Design Principles

The Amoeba Distributed System was designed by Mullender and Tanenbaum at the Vrije Universiteit in Amsterdam.[49] Amoeba is an open system, designed to accommodate heterogeneous hardware and software. The Amoeba Kernel, the replicated operating system running in most of the machines on the network, supports process management and interprocess communication. All other services are provided by server programs that execute in user space. A capability mechanism provides protected communication between clients and services and protected access to objects.[48]

The advantages of open systems over the traditional approach are obvious: operating system kernels become smaller and more maintainable, operating system services are no longer in the kernel, making them portable, and allowing multiple, equivalent, but different services to co-exist side by side.

Data base management systems often have their own operating systems, tailored to this particular application, because traditional operating systems provided the wrong functionality.[70, 73] An open operating system, with the right kind of file service, can support data base management efficiently, while integration with other system services is possible. A hierarchy of services, as illustrated by FIGURE 6.1, allows a logical layering of facilities while the development effort can be shared.

The design of the Amoeba File Server was an experiment. We wanted to try to design a layered file system, where replication, concurrency control, and database management would be in different layers. The bottom layer, the *physical layer*, consists of the storage devices: electronic disks, magnetic disks and write-once optical disks. The next layer is the *Block Service*, providing *virtual disk blocks* of various kinds: fast but crash-volatile storage in memory, stable-storage disk blocks, replicated disk blocks with atomic write on all copies simultaneously, etc. The next layer up is the file system, with concurrency control mechanisms for file access, and the top layer, providing the interface to various applications, provides database management services. This paper concentrates on the middle layer: the file system.

The Amoeba File Service is a distributed file service: a request for an operation on a file can go to any one of a number of file server processes where it will be executed. The layered structure is an advantage here; the Block Service already forms an abstraction away from physical storage locations.

But the layered structure of the file system is also a potential bottleneck for performance of great magnitude: A simple query on a tiny database from a client process invokes the database service, which invokes the file service, which invokes the block service, which finds the block on disk. Caching strategies are essential at all levels of the hierarchy to avoid having to descend to the bottom level of the service hierarchy on each client request. However, caches and concurrency control mechanisms are likely to become enemies: the administration of the caches in a rapidly changing environment can cause more inefficiency than

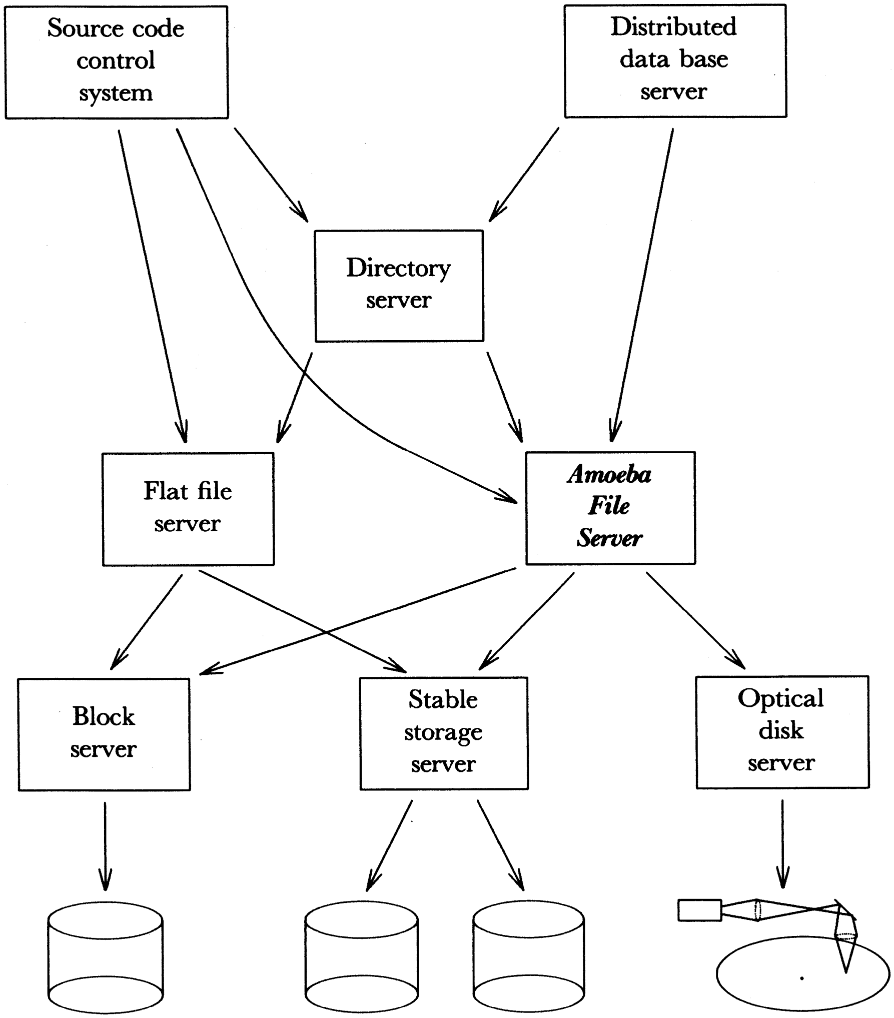


FIGURE 6.1. An example of a storage services hierarchy in an open system.

not keeping caches at all. Obviously, thinking about caching possibilities and strategies have to be an essential part of the design process.

File services must provide the tools for the efficient implementation of as wide a set of applications as possible. This can be realised, in part, by providing a large set of different file services, each tailored for a particular application, but, naturally, it is best to have as few as possible different file services that cover the needs of every conceivable application.

Currently, Amoeba has three file servers: a simple one, written as a student programming project, which implements simple flat files, without concurrency control; a UNIX-like file server, which is used in combination with a UNIX emulation package for running UNIX software on Amoeba; and the Amoeba File Server, described in this paper.

An important design principle was also: 'You should not have to pay for those features you do not need'. A file server, for instance, that implements atomic update on replicated files is a very nice thing to have, but a user who wants to store the output of a compiler, prior to calling a linking loader doesn't share that output with any other user; he is not interested in having his file replicated across five different network nodes for increased availability, nor is he interested in having his file atomicly updated. All such a user wants is a temporary file that can be quickly accessed and changed, and just reliable enough that usually he doesn't need to compile his program all over because the file was lost. On the one hand, our file server should cater for users who just want a reasonably reliable repository for their files, cheap and fast, while on the other hand, other users should be taken into account who need ultra-reliable storage for their files, fancy synchronisation of access by many simultaneous users, and guaranteed availability, who will be prepared to accept that it must be more expensive and slower.

Another important issue in file service design is that the semantics of file service be easy to understand. The interface to the file server must not only be simple, with as few commands as possible, clients must also have a simple conception of the structure of a file, and how to use the mechanisms provided. Even if clients want highly sophisticated things done, like changing a heavily shared file atomically, they should not be burdened with the details of a five step locking protocol, or have to know just how many times the file is replicated.

It is a design goal that the distributed file server should be suitable for an Amoeba environment, using the protection provided by Amoeba's ports and capabilities.[48]

6.1.1. Related Work

Since the beginning of distributed computing, many file servers have been built. In this section we shall look at some that are closely related to our work: XDFS[71] FELIX,[24] SWALLOW,[55] and ALPINE.[7] They all have mechanisms for concurrency control. Most file servers, including the Cambridge File Server,[18] XDFS, FELIX and ALPINE use *locking*,[21] while some, among them SWALLOW, use *timestamps*.[54]

XDFS is a distributed file server that uses the notion of *transactions*. *Open transaction* and *close transaction* commands bracket a series of read write commands to one or more files, and the system guarantees the *atomic property* for these transactions; that is, either all of the changes will be done, and the transaction succeeds, or none, and the transaction fails. XDFS realises the atomic property via so-called *intentions lists*, a list of changes to the file and a two-phase commit protocol.

XDFS uses an interesting locking mechanism to guarantee serialisability: there are three kinds of locks, read locks, intention-write locks, and commit locks. When a client has locked a datum on a server for some time, a timer expires and the lock becomes *vulnerable*. Another client, waiting on that lock, can then prod the server, requesting it to release its lock. If it is in a state to do so, it releases its lock, otherwise it ignores the prod.

The FELIX file server also uses locking, although here it is at the file level. The FELIX locking mechanism is combined with a *version* mechanism: when a file is examined or modified, a new version of the file is created. A new version is created by making a (virtual) copy of the current version; this new version can then be read and modified, and, when all changes have been made, the new version may become the new current version. Sharing is controlled using locks, providing six access modes. Files are tree-structured. A new version is created by copying a pointer to the root of the current version. When it is modified, a copy-on-write mechanism is used which leaves the current version intact. With this mechanism, only the changes between versions are stored.

ALPINE offers the user a choice between locking at the file level or at the page level. File locking is the default, but sophisticated applications are provided with mechanisms for setting and releasing various types of locks on individual pages of a file. A transaction log is kept to enable recovery from failures and deadlocks caused by conflicting locking operations. Brown *et. al.* claim that transaction logs can be implemented more efficiently than a shadow-page mechanism.[7] A transaction log mechanism, however, makes it more difficult to implement an efficient and simple caching mechanism, as shown in § 6.3.5.

Like FELIX, SWALLOW also uses a version mechanism, but the synchronisation of concurrent access is quite different. SWALLOW uses a timestamp mechanism, based on Reed's notion of *pseudo time*. This mechanism is used to ensure the atomic property of updates to collections of arbitrary objects (*e.g.*, files). Additionally, versions do not overlap; that is, they do not share the unmodified portions of the file.

6.1.2. The Amoeba File Service Compared With Other File Servers

The Amoeba File Server is a file server, with a version mechanism similar to that of FELIX, but in contrast to other file servers, it uses a combination of locking[21] with an optimistic concurrency control mechanism.[33, 59, 62] Optimistic concurrency control mechanisms have been used in data base management systems, but we have never seen them used in a file server. Yet, an optimistic concurrency control mechanism, combined with a version mechanism provide a number of advantages, not present in other file systems.

The most important characteristic of a version mechanism, is that the file system is always in a consistent state. Most file systems, update files in place and need a mechanism for bringing back the file system to a consistent state after a crash of a server and possibly also after a crash of a client. A client crash can cause parts of the file system to be inaccessible for some time, for instance, because a rollback operation must be done first to bring the file system back to a consistent state. In the Amoeba File Service, the file system is always in a consistent state (assuming the updates themselves are internally consistent). Server crashes have no serious consequences: there is no rollback, clients need only redo the update that remained unfinished because of the crash. Clients do not have to wait until the server is restored, because they can use another server to do their updates.

In a way, optimistic concurrency control and locking are complementary mechanisms: Optimistic concurrency control maximises concurrency and works best when updates are small and the likelyhood that an item is the subject of two simultaneous updates is small. Locking, in contrast, does not allow as much concurrency, and is more suitable when updates are large and unwieldy and when the probability of an item being subject to more than one update is significant. The Amoeba File Service combines locking and optimistic concurrency control in such a way that updates of large bodies of data (several files) use locking to prevent having to redo them if they clash with another update. Updates of small bodies of data (one file) are less likely to clash with other updates, so an optimistic approach is used here. When necessary, a *soft-locking* scheme can be used in addition to optimistic concurrency control to ward off potential conflicting updates. In all cases, the mechanisms for carrying out updates guarantee consistency of the file system at all times.

The Amoeba File Service provides the necessary mechanisms to maintain caches of data. Caching is an important concept in distributed systems;[37] ITC[61] and CFS,[63] mention caching mechanisms as important parts of the system. Both Amoeba File Servers and their clients can hold data in a cache. In many file systems, it is difficult or impossible to maintain caches, because the integrity of the data in the cache cannot be assured. ITC was not designed for database applications and does not provide complicated machinery for concurrency control; maintaining a cache is relatively simple there. XDFS uses 'unsolicited messages' to tell clients to unlock cached data when it is going to be modified. This makes their caching strategy efficient only for data that is rarely modified. In CFS, shared files do not change after creation which makes caching trivial; a version mechanism embedded in the naming mechanism is used to reflect change.

On the Amoeba File Service, the integrity of the cache need only be checked at the start of a transaction. The cost of checking whether the cache is up-to-date is small, even for files that are frequently modified. Furthermore, the Amoeba File Service needs no unexpected 'unsolicited messages.'

6.2. The Block Server

The principle of separating the issues of file service and block service makes it easy to combine different methods of storage (*e.g.*, stable storage [35]), and storage media (*e.g.*, small fast 'electronic disks,' large slow magnetic disks, very large optical disks) in one system. Carefully designed, disk service can combine high speed with high reliability, using techniques, such as caching and dual storage, both on fast, but not so reliable storage, and slow, but very reliable storage.

We assume the block service implements as a minimum commands to allocate, deallocate, read and write fixed size blocks of data. Protection must be provided, so that a block, allocated by user *A* cannot be accessed by user *B* without *A* 's permission. Writing a block must be an atomic action, with an acknowledgement that is returned *after* the block has been stored on disk. This property is vital for the implementation of atomic update on files.

The block server can implement a simple locking facility on individual blocks. Based on this, file services can realise concurrency control policies. The Amoeba File Service, to commit a version of a file, for instance, will exclusively lock and read a block, examine and modify it, then write and unlock the block again.

Magnetic disks and optical disks do not usually lose their information in a crash, but it does happen occasionally. In any case, they are at least temporarily inaccessible. In order to achieve high availability in the face of disk crashes, it is necessary to store every block at least twice, on different disks, managed by different servers. Lampson and Sturgis[35] have suggested a method to use dual disk drives to implement *stable storage*. With minor modifications their method can be used to provide disk service which continues to be available when single-site crashes occur.

6.3. Amoeba File Service

The Amoeba File Service was developed for, but is not restricted to, the Amoeba Distributed Operating System. It implements the file system as a tree of pages, whose subtrees are files, and uses a combination of an optimistic concurrency control mechanism and a locking mechanism to prevent conflicts in simultaneous updates.

The Amoeba File Service implements optimistic concurrency control by a version mechanism: When a client opens a file for modification, a new version of the file is created, which initially behaves like a copy of the file. Then the modifications are made, and finally a *commit* operation makes the modifications permanent by replacing the previous current version with the new one. After commit, a version becomes immutable. Several uncommitted versions of the same file can exist at a time. The Amoeba File Service checks on commit whether the modifications to the file constitute a serialisability conflict (see [33]).

The current state of a file is contained in the **current version**. **Committed versions** represent past states of a file; **uncommitted versions** represent possible future states of the file. Files are accessed by their **file capability**, versions by their **version capability**. Atomic updates on files are bracketed by creating a version and committing a version. The current state of a file is always represented by the contents of the current version. Committing a version makes that version the current one.

The Amoeba File Service is a distributed service. Several server processes can be established on one or several physical machines, and each server is capable of handling updates on any file. Each version has a **manager**, the server process which created the version. Different versions of a file can have different managers. A client will typically direct all requests to the Amoeba File Service to a server process that is close to it. New versions will thus tend to be close to the clients that ordered their creation.

A version is represented as a tree of **pages**. Clients can read or write a page at a time. The maximum length of a page is determined by the maximum length of an Amoeba message transaction: 32K bytes. This ensures that pages can be read and written in one (atomic) action.* A page may contain both data and references to pages further down in the tree. A reference consists of a block number and some flag bits that Amoeba File Service uses for concurrency control. The number of data bytes in a page is variable (per page) up to the maximum size of a page.

Clients have explicit control over the shape of the page tree. Pages within a file are referred to by a *pathname* which is constructed as follows: The root page has an empty pathname. The pathname of a page that is not the root is the concatenation of the pathname of its parent page with the *index* of its reference in the array of references in the parent page.

This file representation has been chosen with the express intention of giving clients (file systems, data base systems, source code control systems, etc.) as much control over the shape of files as possible. Using the file structure provided by the Amoeba File Service, objects ranging from linear files to *B-trees* can easily be represented.

The Amoeba File Service provides a set of commands for the management of files and versions. There are commands to read and write the pages of a version and commands to manipulate the shape of a version's page tree (split pages into two, move subtrees to another part of the tree, etc.).

Files can be grouped together in *'superfiles,'* and superfiles can be grouped in other superfiles. Such a superfile structure is also a tree structure. A superfile is, in fact, almost exactly like an ordinary file: All pages of a superfile may contain data, exactly like an ordinary file; the root page of a superfile, however, contains

* Arbitrarily long pages can be written atomically by writing them back-to-front as a linked list, whereby the head block is (over)written last, and the other blocks in the list are allocated from the pool of free disk blocks. After writing, the blocks making up the previous linked list can be freed.

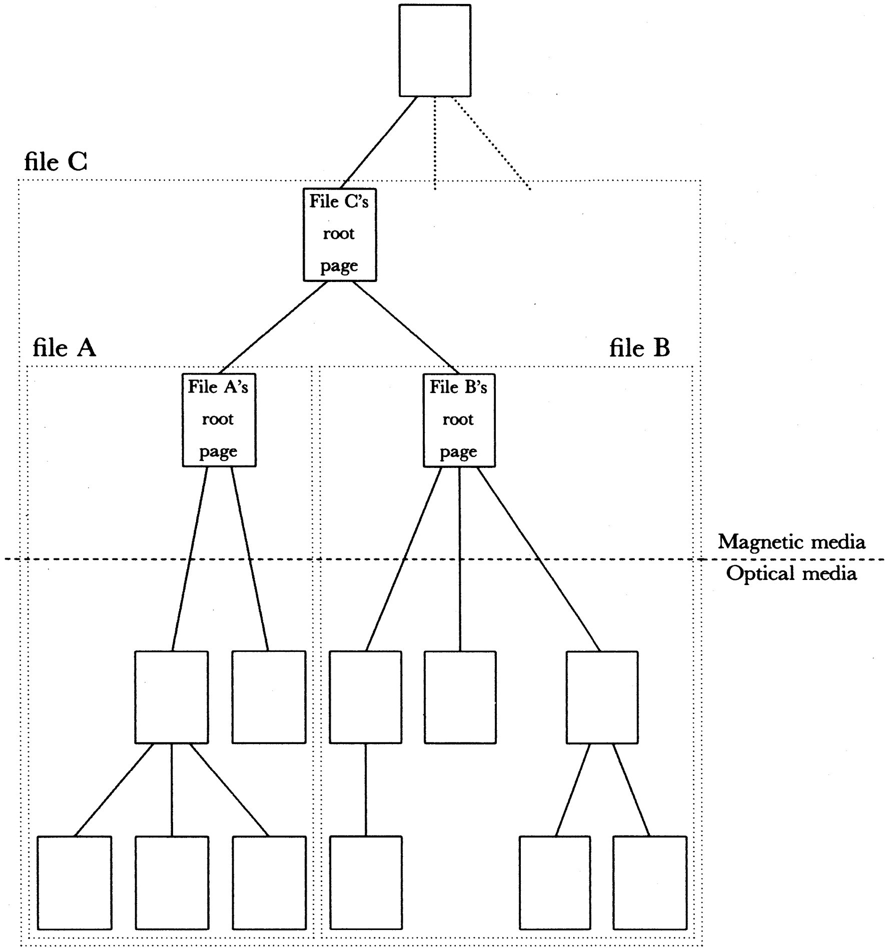


FIGURE 6.2. A file has the structure of a tree of pages. A superfile can be viewed as a tree of files or also as a tree of pages.

references to the root pages of other files, superfiles, or both. The Amoeba File Server provides atomic update on files, or superfiles. Files or superfiles without a common root cannot be updated atomically.

The top of the tree, that is, the collection of root pages of files, is stored on magnetic random-access media, for instance, such as provided by the *stable-storage* server, mentioned in the previous section. The lower parts of the tree, that is, the collection of non-root pages of files, can be stored either on magnetic disk, or write-once media, such as optical disk. As illustrated in FIGURE 6.2, a subtree, whose root is in the upper part of the tree, *e.g.*, *file A*, can be viewed as a file; it can be modified atomically using the methods described below. Amoeba files, unlike files in most file systems, thus form a nested structure: A subtree whose root page is inside another subtree may be viewed as a file within another file. *File A* and *file B*, for instance, are both subfiles of *file C*.

6.3.1. File Representation

A file is a collection of versions, ordered in time. When a new version is created, it behaves as if it were a copy of the current version. In fact, when it is created, a new version shares its page tree with the current version, and only when a page is changed is the page duplicated. The Amoeba File Service file representation is therefore a differential file representation, similar to that of Felix.

Pages are stored by the block server in such a way that they can be read and written as atomic actions. Associated with each page is a small header area that the Amoeba File Service uses for administrative purposes.

The root of a page tree is referred to as the **version page**. The client data in a page has no predefined structure. Clients are free to write them as they see fit. The references in a page to pages further down the tree are for internal use by the Amoeba File Service and can only be read and written by servers.

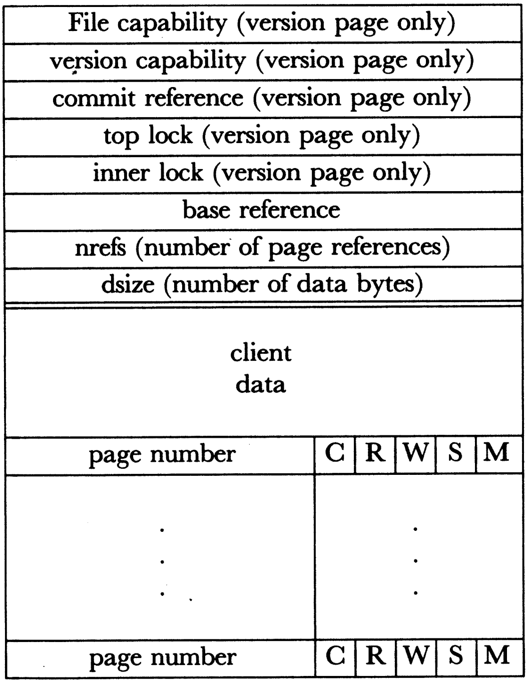

File capability (version page only)
version capability (version page only)
commit reference (version page only)
top lock (version page only)
inner lock (version page only)
base reference
nrefs (number of page references)
dsize (number of data bytes)
client data
page number
C R W S M
page number
C R W S M

Figure 6.3. The Amoeba File Service page layout

The layout of a version page is shown in FIGURE 6.3. The layout of internal pages is the same, with the exception of the first five fields, which are not used. Each page is divided in two areas, a header area and the page itself; the separation is indicated by the double line. The first field in the header area of a version page is the *file capability*. This field gives the capability of the file whose root the version page is. The next field is the *version capability*, the version of the file whose root the version page is. The *commit reference* field is also used in version pages only; its use will be explained presently. The *top lock* and *inner lock* are used to tell whether a page is currently involved in an update of a superfile whose root is higher in the page tree; their function will be explained in a later section. The fields mentioned just now are only present in a version page. They are absent (or ignored) in other pages. The remaining fields, to be mentioned below are present and used in all pages, root pages and internal pages alike.

The *base reference* field, present in all pages of a version, is the block number of the page that this page was based on (copied from). The *nrefs* field holds the number of page references this page contains. If this field is zero, the page is a leaf page. The *dsize* field gives the number of data bytes. The page itself contains the *reference table*, with an entry for each child page, and the data area where the client data is kept.

The reference table is an array of *page references*, which contain a *block number*, and five flags, C, R, W, S, and M. The page reference points to a page in the next level of the page tree, the C flag, when set, indicates that the page was copied and is no longer shared with the version it was based on. The R flag indicates whether the data of that page has been read (it is needed to decide if an uncommitted version may be committed as explained in § 6.3.3), the W flag indicates whether the data in the page was written (changed), the S flag tells if the references have been used (searched), and the M flag indicates whether the references were modified. As we shall see, it is not possible to access a page without copying it, nor is it possible to modify the references without looking at them. This reduces the number of flag combinations to 13, which allows encoding the flags in four bits. Amoeba uses 28 bits for a block number and four bits for the flags.

Pages are accessed from their parent page by the *index* in the reference table. An arbitrary page in a version can thus be accessed from the root by indexing into the reference tables of several pages starting at the root (version page) of the page tree. Pages thus have path names consisting of a string of n-bit numbers.

A file is made up of a sequence of committed versions and possibly a collection of uncommitted versions. The version pages of the committed versions form a doubly linked list. Each committed version's base reference points to the version it was based on (its predecessor) and its commit reference points to the next committed version. The current version's commit reference and the oldest version's base reference are nil. The uncommitted versions are attached to the list through their base references, which point to the version they were based on; note that this is always a committed version. A typical file could look like the

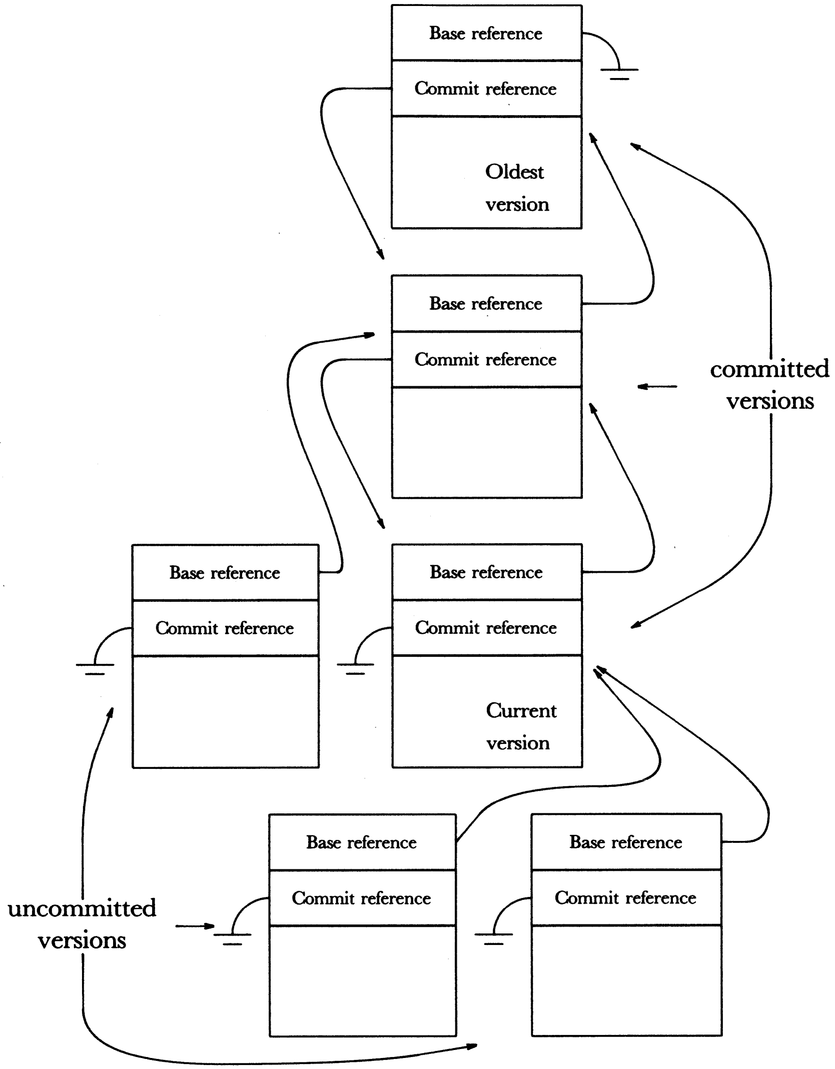


Figure 6.4. The *'family tree'* of a typical file. Only the version pages are shown. The page trees descending from the version pages are not shown.

one in Figure 6.4, where we have just shown the version pages and their base and commit references.

6.3.2. The Copy-on-write Mechanism

In the next section we shall discuss the mechanisms that are used to implement atomic update and guarantee serialisability, but before we go into that subject, a proper understanding of the copy-on-write mechanism and the R, W, S and M flags in the page table is needed.

The R, W, S and M flags are needed primarily for deciding about committing versions. In order to be able to serialise two simultaneous updates to a file, the Amoeba File Service must know which parts of the file were read and which parts were changed (written). When set, the R flag indicates that the data in the referred-to page was read. The W flag indicates its data was written. The two flags operate independently of one another. The S flag tells that the references have been searched, the M flag tells that the references have been changed. These flags are not independent. When the M flag is on, the S flag must also be on; it is not possible to modify the references without consulting them.

When a page is read, the pages on the path to it must be searched. This implies that, if a page has not been searched, the subtree of which it is the root cannot have been searched or read either. Hence, a cleared S flag indicates that the descendants of the referred to page have not yet been accessed.

For writing pages in a version, a 'copy-on-write' mechanism is used. When a page is written, a new block is allocated for it, leaving the old page intact. Then the page reference in its parent page is updated to point to the newly allocated page and its W flag is set. This changes that page, however, and, if it is still shared with another version (i.e., it hasn't been copied-on-write yet), this change must also be made by allocating a new block for it and writing the new contents of the page to that new block. Every change thus bubbles up from the leaves of the page tree to the root page. The root page—the version page—is the only page that is always written in place, because it is never shared with another version. When a non-root page is thus copied, the C flag is set in the reference to it (in the parent page). A page is thus only copied once; after it has been copied for writing, it can be written in place when it is written again.

It is clear now that, when a page has not been copied, its descendants can not have been copied either. Hence, a cleared C flag in a page reference indicates that the referred to page and all its descendants have not (yet) been copied, but a set C flag only indicates that the referred to page was copied. Like the S flag, it does not show whether its descendants have been copied.

A similar mechanism does not exist for the R, W and M flags. When a page is written, it and the pages between it and the root of the page tree must be copied, but the parent page of a written page is not considered written or modified, although, strictly speaking, it has changed. A parent page is only considered written if it's client data was written, and modified if pages were added or deleted.

Page trees are usually partially shared between versions. This implies that the flags indicating access to pages are also shared, even though these pages have been accessed in different ways in different versions. This presents no problem, because the serialisability test need not descend shared parts of the page tree since they have not been accessed.

The flags, indicating whether a page has been read, written, modified or copied are stored in its parent page in the page tree; the root page is therefore the only page that does not have associated C, R, W, S and M flags in the file tree to indicate if it was copied, read, written, searched or modified. The managing server keeps these flags separate. The root page is always copied, by the way.

When a page is first read, the C, R, W, S and M flags it contains for its child pages must be initialised to zero. This requires changing that page. The Amoeba File Service must therefore not only shadow pages that were written, but also pages whose descendants were read. As we shall see later, once a version has successfully committed, the information contained in the R and S flags is no longer needed. The Amoeba File Service garbage collector may remove pages that were copied but not written or modified and reshare the corresponding page from the version on which it was based.

6.3.3. The Optimistic Concurrency Control Mechanism

As long as updates are carried out one after the other, commit always succeeds and requires virtually no processing at all. When two or more updates proceed concurrently, however, the server must check whether commit can be allowed by testing whether those updates can be serialised. If so, the commit is allowed; if not, failure is reported to the client, and the client must redo the update.

When there is no concurrency, a new update will not start until the previous one has been finished; that is, a new version will not be created until the previous version has been committed. The next version is thus always based on the previous one. Updates are concurrent when a new version is created while another, uncommitted version still exists. This implies that concurrent updates are based (sometimes indirectly) on a common (committed) version.

Kung and Robinson in their paper on optimistic concurrency control divide file update into three phases: the read phase, the validation phase, and the write phase.[33] The validation phase checks *serial equivalence* of transactions T_i and T_j by testing whether one of the following conditions hold:

(1) T_i completes its write phase before T_j starts its read phase.

(2) The write set of T_i does not intersect the read set of T_j, and T_i completes its write phase before T_j starts its write phase.

(3) The write set of T_i does not intersect the read set *or* the write set of T_j, and T_i completes its read phase before T_j completes its read phase.

If one of these conditions hold, the effect of updates T_i and T_j is the same as when T_i had finished before T_j started.

The Amoeba File Service carries out updates in such a way that the critical section of the validation phase and the complete write phase are consist of one atomic action. This implies that the write phases of two transactions can never overlap and the serialisability test for two updates in the Amoeba File Service reduces to

(1) Version *V.i* is committed before version *V.j* is created.

(2) The write set of version *V.i* does not intersect the read set of version *V.j*, and *V.i* is committed before *V.j*.

The Amoeba File Service carries out its validation test when a client process requests a version to be committed (*i.e.*, when the client process signals the end of a transaction). In the test, it is only necessary to check if serialisability conflicts will occur with versions that have already committed. In principle, the commit mechanism works as follows.

The check whether condition (1) holds, and if it holds, the write phase, are carried out as one atomic operation, described below. If condition (1) does not hold, a test has to be made whether condition (2) holds.

When a client requests to commit a version, *V.b*, that is based on the current version, *V.a*, condition obviously (1) holds, because *V.b* was created after *V.a* was committed. Therefore, the Amoeba File Service allows all commits of versions based on the current version. The mechanism for this is demonstrated in FIGURE 6.5.

Let us assume client *C* sends a request to commit version *V.b*, which is based on version *V.a* to *V.b*'s managing server, *M.b*. Server *M.b* then proceeds as follows. First it ascertains that all of *V.b*'s pages are safely on disk. Then it sends a **set commit reference** request to *M.a*, the manager of *V.a*, the version that *V.b* was based on. (*V.a* is specified in the *base reference* field of *V.b*'s version page.) *M.a* must then do the following without allowing other requests to interfere. First it must check whether *V.a* is still the current version. If so, there is no conflict and the commit is carried out. The check for currentness is simply performed by examining *V.a*'s commit reference. If it is nil, *V.a* is the current version, and the commit reference is set to the page number of *V.b*'s version page. This makes *V.b* the current version, and automatically the updates made to *V.b* are made permanent.

The test and set the commit reference is the only critical section in version commit. In order to make it an indivisible action, only one server may be allowed to read the version block, test the commit reference, set it, and write it back. If the disk server implements a test-and-set operation, any server can be allowed to carry out a commit.

FIGURE 6.5(a) shows the situation before commit, FIGURE 6.5(b) after the commit has successfully been carried out. *M.a* returns an acknowledgement to *M.b* and *M.b*, in turn, returns an acknowledgement to *C*.

Let us now examine the case where *V.a* is no longer the current version, which means that another update, concurrent with that of *V.b*, has taken place and was committed. Let us assume the situation of FIGURE 6.6; *C* sends a request to *M.b* to commit *V.b*. However, *V.c* is now the current version, also based on *V.a*. First, *M.b* proceeds as before, and sends a **set commit request** to *M.a*; only this

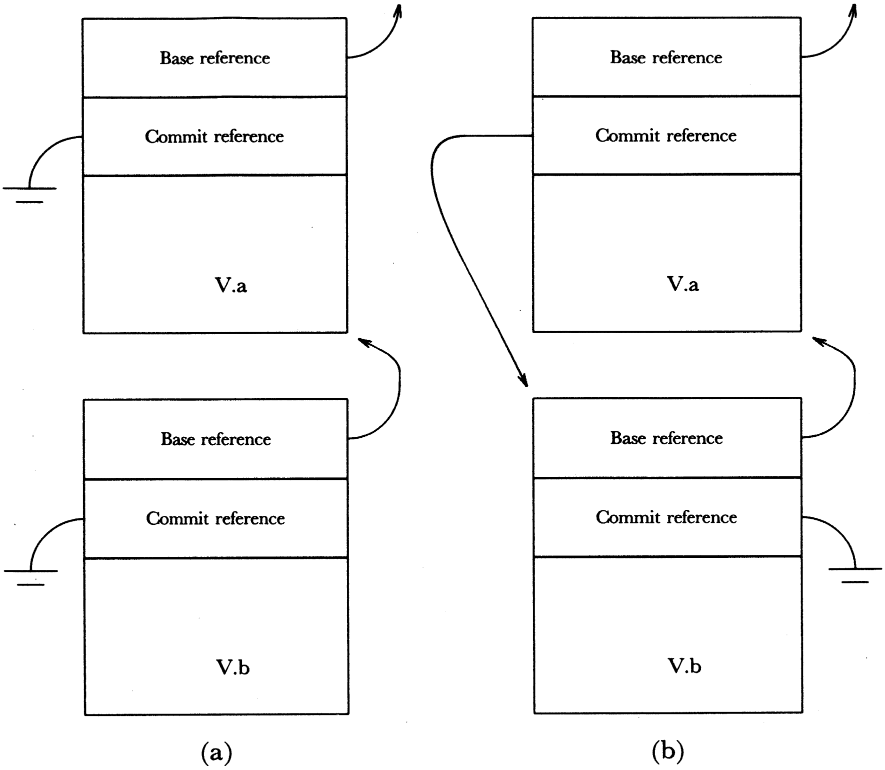


FIGURE 6.5. *V.b* succeeds *V.a* as the current version. (a) shows the situation before the commit, (b) shows the situation after the commit.

time, discovering *V.a*'s commit reference is already set, *M.a* does not carry out the commit, but returns *V.a*'s commit reference instead. This is the block number of *V.c*'s version page.

M.b must now check if the concurrent updates of *V.b* and *V.c* are serialisable; that is, test whether condition (2) holds. *V.c* has already committed, so if the two updates are serialisable, *V.b* must come after *V.c*. This implies that there must be no overlap of *V.c*'s write set (the pages written during the update of *V.c*) and *V.b*'s read set (the pages read during the update of *V.b*). Since *M.b* received the block number of *V.c*'s version page, it can descend *V.c*'s and *V.b*'s page trees in parallel to examine if there is a serialisability conflict. This is tested using the *R*, *W*, *S*, *M*, and *C* flags in the page references. Note that unshadowed parts of the tree in either *V.b* or *V.c* need not be visited since they haven't been accessed.

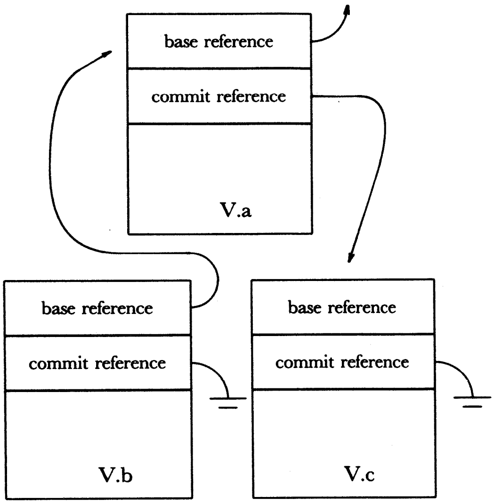


FIGURE 6.6. *V.b* wants to commit, but is no longer a descendant of the current version, *V.c*.

While descending the two page trees, checking the serialisability constraint, *M.b* also prepares the new current version, which must combine the updates made in *V.c* with those made in *V.b*. This is done by replacing unaccessed parts in *V.b*'s page tree by corresponding written parts in *V.c*'s page tree.

Both the serialisability test and the combination of the changes made by two concurrent updates are made in one pass over the page tree. Unvisited branches in either page tree are not descended, which makes the serialisability check quite fast when at least one of the concurrent updates is small.

An important property of the serialisability test is that it can be carried out in parallel with other updates of the file. While the routine *serialise* descends *V.b*'s and *V.c*'s page tree, other versions are allowed to commit, and other serialisability tests can also be carried out.

If *serialise* returns TRUE, *V.b* is ready to become *V.c*'s successor as the current version, and a *set commit reference* command is sent to *V.c*'s manager. If *V.c* is still current, this succeeds; if not, the serialisability test is repeated for *V.c*'s successor. This repeats until either the *set commit reference* command succeeds or *serialise* returns FALSE.

In the latter case, when *serialise* returns FALSE, the concurrent updates are not serialisable, and *V.b* is removed, and its owner notified. The update can be retried on another version.

6.3.4. The Locking Mechanism

In the previous section we have described the update mechanism for a single file. In this section we describe the mechanisms for updating superfiles which may contain several smaller files.

Before continuing, some terms are defined to simplify discussions. The upper part of the tree, which contains the version pages for the files in the system, will be called the **system tree**. A file whose root is a leaf of the system tree, i.e., an ordinary file, will be called a **small file**, although a 'small file' may, of course, be arbitrarily large. In FIGURE 2, for instance, file A and file B are small files. A file whose root is not a leaf node of the system tree will be called a **super-file**. In FIGURE 2, file C is a super-file. A small file or super-file whose root is contained in a super-file will be a **sub-file** of the super-file. A tree that makes up a small file or super-file is a **page tree**.

Updates of small files still use the optimistic method for update: Two updates on different small files do not interfere with each other since they affect disjoint page trees. Two updates of the same small file use optimistic concurrency control, as described in the previous section, to maintain integrity.

Updates of super-files, however, must use different rules. Updates on super-files generally require larger amounts of processing and affect more pages than updates on small files. Consequently, the likelyhood of a serialisability conflict is greater for updates on super-files. Additionally, the work lost because of a serialisability conflict is usually more in the case of super-file updates.

For these updates *locking* provides a better form of concurrency control, because it warns in advance that two updates are likely to cause a conflict. Locking has the drawback, however, that after a crash, locks have to be cleared before the system can resume operations. We deemed it a challenge to find a locking mechanism that requires no special recovery in case of crashes. Our method is described below.

Each version page contains two *lock* fields, the *top lock* field, and the *inner lock* field. A file is considered to be *locked* if the lock field is non-zero. Locks only have meaning in the current version. We assume it is possible to test the two lock fields for zero and set one of them in one atomic operation.

When an update is made to a super-file, the *top lock* is set in its version block, and the *inner locks* are set in visited internal nodes of the file tree that are version blocks of sub-files. When an update is made to a small file, the *top lock* is also set in its version block, but, since small files have no internal version blocks, no *inner locks* have to be set.

Updates on super-files happen in exactly the same way as updates on small files, with the exception that locks have to be checked and set while the update is in progress. As in the case of small files, a version must also be created for a super-file before updates can be made. Before a version may be created, however, the version block for the current version must be locked.

The algorithm for creating a version is the following: If the file is a super-file, check the *inner lock* and *top lock* fileds, and, if they are both zero, set the *top lock*.

If one of them is non-zero, wait until it is cleared, then try again. (The waiting process will be described later; locks contain the name of the locking server, which is used to realise an automatic warning mechanism for waiting updates.) If the file is a small file, only the *inner lock* must be tested, but the *top lock* set. Thus, a small file can be subject to more than one update at the same time, using the optimistic method of concurrency control. When multiple, concurrent updates are allowed on a super-file, this rule can be used on super-files as well.

Assume, for instance, that an update of file A in FIGURE 2 has to be carried out. It is a small file, so only its *top lock* will be set. Other updates on file A can proceed concurrently: the *inner lock*, which is not set, is tested, and concurrent updates can be carried out as described in the previous section.

If an update, while descending the page tree, discovers a *top lock*, it must wait until the lock is cleared before that subtree can be entered. It is not possible to encounter an *inner lock* while descending the page tree.

Suppose again that file A is being updated, so its top lock is set. An update of file C can proceed, as long as its left subtree, which is file A, is left untouched. When C's left subtree is descended, however, A's top lock will be encountered, and C's update must wait until A has been committed and its lock has been cleared.

The use of the inner locks will become clear when we assume an update on file C descends A's page tree. This update will cause A's inner lock to be set. When an attempt is now made to update A, the inner lock will be encountered, and the update must wait until it is cleared.

The commit operation is somewhat more complicated for super-files than for small files. Commit on a small file or a super-file works as described in the previous section. However, commit on a super-file is not finished when the *commit reference* is set. After commit on a super-file, the page tree must be descended to commit the sub-files of the super-file, and clear the locks. These commits always succeed, because the locks prevent access by other clients during the update to the super-file.

It is not difficult to see that this locking mechanism gives exclusive access to any subtree of the file system, and therefore provides a concurrency control mechanism. It can also be seen that sub-files, not accessed by an update, are not locked and therefore accessible to other updates. Full concurrent update remains possible on small files, because simultaneous updates on the same small file need not wait for *top locks*.

However, it is possible to use *top locks* on small files as hints which indicate that the file is likely to change soon. An update, known to affect large parts of a small file, can thus be postponed until the file is 'idle.' In contrast to this *soft locking* scheme, it is also possible to allow more concurrency on updates of super-files. The rules for creating a version may be relaxed to allow creating a version when the version block's *top lock* is set. The optimistic concurrency control which still lurks underneath this locking mechanism will see to it that no harm is done 'concurrencywise.'

When a server process crashes in the middle of an update, no harm is done to the integrity of the file system; the optimistic method underneath sees to that. The locks remain, however, rendering some files inaccessible. Fortunately, the mechanism described above for waiting on locks also provides a mechanism for crash recovery: When the server crashes, the outstanding transactions with the server crash as well, telling all servers waiting on locks that the process holding the locks has crashed.

A server, waiting on a *top lock* proceeds as follows: If the commit reference is off, the lock can be cleared without further ado, and, when the page tree is descended, *inner locks* (containing the same server name, of course) can be cleared or ignored. If the commit reference is set, the version it refers to is current. The version with the lock, and the current version are traversed simultaneously, and the commit references of the sub-files are set, finishing the work of the crashed server. A server waiting on an *inner lock* ascends the *system tree* to the first page without an inner lock, or a page with a *top lock*. If the page thus found has no lock at all, the *inner lock* that the server was waiting on can be ignored. If the page found has a top lock, it is treated as described above.

6.3.5. Maintaining a Cache

An important form of optimisation is caching. It is a defect in most distributed file systems that it is virtually impossible to keep local copies of remote data around, because of the difficulties of keeping the local copies up-to-date. The decreasing cost of primary memory makes caching techniques increasingly useful both for file servers and their clients.

The Amoeba File Service—by design—is especially suited for caching. A version, from the moment of its creation, behaves like a private copy of a file that cannot change without the owners consent. Both Amoeba File Servers and their clients can therefore maintain a cache which, for the most recently used versions of a set of files, contains collections of pages. When a client requests a server to create a new version of a file, the client, the server, or both, examine their cache to see if there are any pages of a previous version of the file that can still be used. The mechanism for this is simple, as shown below.

For each file, a client or a server can make a cache entry, consisting of pages of the most recent version it has had locally. When a request for a new version of the file is made, a serialisability test is made between the version used for the cache entry and the current version in order to find out which blocks of the cache are still valid. If the serialisability test succeeds, all blocks are still valid; if not, the blocks that cause the test to fail must be discarded. Note, that it is not necessary to transmit pages while making the serialisability test. If the cache holder is a client, the version capability must be sent to one of the Amoeba File Servers so the serialisability test can be made, and the server returns a list of path names of pages to be discarded. The server responsible for carrying out the test can make the test itself, or it can delegate the task to the server holding the most recent version for efficiency.

If a file is not shared, the cache entry will always be based on the current version. The serialisability test for finding out if the cache entry is up-to-date is then a null test which always succeeds. Even for shared files the page cache can be quite efficient. As shown previously, the serialisability test can be made in time proportional to the size of the intersection of the set of pages of the version in the cache and the union of the sets of pages in the versions since then. The server making the serialisability test likely has parts of the most recent version in its cache, reducing the number of disk accesses and the amount of network traffic further still.

It is worth noting that, in contrast to other file systems, the page cache does not have to be a 'write through' cache: When a page in a version is written, it need not be written to stable storage immediately. This can be postponed until just before commit.

The Amoeba File Servers can also conveniently cache the concurrency control administration, the flag bits. This allows serialisability tests without having to read the page tree. However, the flags must also be present in the files themselves to make crash recovery possible.

6.4. Conclusions

The Amoeba File Service combines a number of concepts from the operating systems' world, the distributed systems' world, and the database world in a novel way. To the best of our knowledge distributed file servers have not been constructed using optimistic concurrency control. Yet, it provides a number of advantages not often encountered in other file systems.

With a version mechanism, the file system is always in a consistent state. After a crash, there is no necessity for recovery: no rollback is required, no locks have to be cleared, no intentions lists have to be carried out. Optimistic concurrency control allows a maximum of concurrency in accessing files. Some updates will have to be redone when concurrent updates are not serialisable, but with the unbounded potential of computing power that distributed systems offer, redoing an operation now and then is acceptable.

Still, starvation may occur, especially when a large update must be carried out on a heavily shared file. The locking mechanism can be used to lock a file when it is known that the update is large, and the probability of a serialisability conflict serious.

The file system should be organised carefully to avoid that updates on superfiles have to occur too frequently. To this end, each small file should be self-contained as much as possible, so most updates will be on small files. This allows a large degree of concurrency. Locking should be the exception rather than the rule.

Page caches can be maintained, both by end-user processes and Amoeba File Server processes. We believe our method is superior to that in XDFS because no unsolicited messages are necessary. These cause an unneeded additional complexity for client processes.

The version mechanism and the page tree closely resemble the mechanisms in FELIX. However, FELIX uses locking at the file level. The idea behind our system of not locking small files is that many updates, even on the same file, do not affect the same parts of the file. For example, changes in an airline reservation system for flights from San Fransisco to Los Angeles do not conflict with changes to reservations on flights from Amsterdam to London.

The Amoeba File Service provides mechanisms that allow both sophisticated and simple applications to use its services efficiently. We have discussed the methods for concurrency control at some length, perhaps creating the impression that simple-minded applications—such as the example, mentioned in the introduction, of a compiler that needs to make temporary files—must once again pay the price of all that complicated machinery for guaranteeing serialisability. This need not be the case at all. Since pages of 32K bytes can be written, one such page is often large enough to contain a whole file. Writing these one-page files is efficient; no concurrency control mechanisms slow it down.

A last advantage of the Amoeba File Service is that it is eminently suitable for a file system on write-once media, such as optical disks. Optical disks show great promise for the future, because of low cost and huge capacity. Traditional file systems are not suitable for these media, because files cannot be overwritten on a write-once device. The version mechanism, coupled with a cache in which uncommitted files are kept until just before commit seems an ideal file store for optical disks.

7

SERVICE ACCOUNTING AND CONTROL

A computer system provides to its users a set of services: executing programs, storing files, printing listings, typesetting Ph.D. theses, etc. Each of these services make demands on the system's resources: computer time, disk space, paper, film, etc. This chapter deals with the registration of the use of services, and how this can be controlled in a distributed environment.

Accounting is the registration of the use of services. Such registration is needed, not only in commercial computer systems for billing clients for services rendered, but also in non-commercial systems for dividing available resources evenly over competing users.

Resource control is the mechanism for enforcing resource management policies. Generally, a client (which can be a human user, a process, or a service) is not allowed an unlimited supply of any resource. Some resources are expensive (*e.g.*, phototypesetter pages), most are available only in limited supply (*e.g.*, disk blocks). Policies for resource management dictate when and how much a client may use of a resource. Resource control mechanisms realise these policies.

Resource control, however, is just a special area of **service control**: in the ***Amoeba*** distributed operating system all resources are made available through services, defining the allowed operations on the resource, but also defining the rules for its accessibility. The traditional operating system view of accounting resources is limited, because more abstract entities, such as the expert knowledge, embedded in a particular program, are not viewed as resources, and hence not accountable. The more general view of building mechanisms to account for the use of services, rather than resources, is obviously preferable.

Accounting and service control are closely related. Service control is not possible without accounting mechanisms; therefore accounting mechanisms must be designed with regard to both existing service management policies and desirable, not-yet-existing resource management policies. Both accounting and service control mechanisms must be designed with great flexibility to allow them to be used for carrying out a variety of existing policies, but also for many as yet unthought of policies of the future.

7.1. Accounting and Service Control

In this section we show the differences between accounting and resource control mechanisms used in traditional centralised operating systems, and those necessary for open distributed operating systems as we envision them.

7.1.1. Closed Centralised Systems

Until a few years ago, a typical computer centre had one or two huge computers with many peripherals attached to them. Running the computer was expensive, and everything was oriented towards using it as efficiently as possible. This way of thinking is clearly illustrated by the accounting mechanisms used by most computer centres: the unit of accounting is a 'system second.' As the name suggests, a system second indicates one second of CPU time, although it is not solely used as a unit of CPU time consumed, it is also used to account for i/o done, as a measure for printed pages, for the use of tape drives, etc.

Resource control policies consist mostly of giving users a **budget**, an amount of system seconds with which a user buys cpu seconds, disk space, etc. Periodically, an amount can be added to the budget, the amount often based on past use and estimated needs. Additionally, for each type of resource there will be a maximum, independent of the current budget: time limits on processes, quota on disk space, so many pages of output, etc.

More instructive than summing up what traditional accounting mechanisms can do is, perhaps, to look at some of the things traditional accounting mechanisms do not do. There is no *software accounting*, accounting of who uses how much of which software. In the past this was not necessary. The programs used by clients of the computer centre would either be owned by the computer centre or by the clients themselves. The computer centre could obtain the development cost of the software, or the cost of buying it out of the *system seconds* income. Software was cheap compared to hardware in the early days of computing, and this is reflected in the accounting and resource control mechanisms of many computer centres to this day.

Another useful feature lacking in the accounting mechanisms of traditional systems is a mechanism that one user of the system can use for accounting services rendered to another user. The traditional view has always been of one producer of service, 'the system,' and many consumers, 'the users.' This view, by the way, is not only visible in the structure of accounting mechanisms, but also in the structure of other facilities, such as the file system, which often prevents users from sharing files, or from running each other's programs.

7.1.2. Open Distributed Systems

There are important differences between traditional closed centralised systems and modern open distributed systems that affect the mechanisms needed for accounting and resource control. Whereas in traditional systems accounting was primarily done to register consumption of *resources*, in modern open systems it is done to register the consumption of *services*. The distinction is moot, perhaps, but it illustrates a change of view of operating systems, away from just disk blocks, cpu seconds and phototypesetter pages, towards an integrated view of services in a much wider sense of the word. A service is not just the provision of disk space or computing power, but it is these things, combined with the research effort, the development, the maintenance of the service.

To illustrate how this integrated view leads to fairer accounting policies, consider two compilers, a good one and a bad one. The latter will be slower, will produce worse code, and it will produce less comprehensible error messages. Traditional accounting methods, however, will register more resource consumption using the bad (slow) compiler, hence the users of the system will be charged more for using the worse service. This is not necessary when accounting policies are not based on consumption of resources, but on both quantity and quality of service, rather than just quantity of resources.

Another development in open distributed system is that services are no longer exclusively provided by 'the system.' The notion of one central authority providing service to many users no longer applies in distributed systems, consisting of a mixture of collectively owned and privately owned processors. But even when 'the system' as a whole is owned by one administration, it may still be desirable to allow ordinary users to provide services in return for financial gain or other services.

A third major change is in the structure of services. In traditional systems accounting was done at one level only, basic resources, such as disk blocks and CPU seconds. Modern, service-oriented systems have a hierarchy of services. One service makes use of other services in order to function. This is clearly illustrated in the file service of the previous chapter, which needs disk service and stable storage service, but also of process service to execute the server processes, boot service to guard against crashes, etc. The accounting mechanisms must provide the tools for accounting in hierarchies of services.

7.1.3. Accounting

Accounting is the registration of service consumption by clients. The problem is to determine *how much* of a service clients use. There are several aspects to this problem. First, the amount of mutual trust needed between service producer and service consumer: can a service claim it gave service to a client, while the client denies having received it? Second, the way in which service consumption is measured: is compiling a ten line program half as cheap as compiling a twenty line program? Third, the relation between different services: how does file service compare to phototypesetter service?

In solving these problems, an important aspect is the relation between the security required and mutual trust needed on one hand, and the cost in effort and complexity of realising such security on the other. At one end of the spectrum, one can imagine a system where clients are trusted to register their resource consumption themselves; a system, thus, that works only through a high degree of mutual trust. At the other end of the spectrum, a system can be imagined, where there is no trust between clients and servers whatever. Clients and servers negotiate contracts, perhaps, establishing a mutually verifyable computation of cost, making transaction logs adorned with digital signatures for examination by an impartial referee, in case of dispute. In most cases, neither approach is acceptable, the first, because it assumes more honesty on the part of clients than can reasonably be expected, the second, because its realisation is far too costly in terms of complexity and speed. A reasonable middle course has to be found, accepted as trustworthy enough for clients and servers to use, and acceptable in cost.

In traditional accounting mechanisms, all service is provided by the 'system,' including accounting. Clients must thus trust the 'system' completely, since it both provides the service and does the accounting. In a distributed system, with many users providing service to others this strategy is not possible: it depends on one central authority providing service and accounting.

In an open system, where every user may decide to set up service for other users, an 'open market' of services is likely to arise. Services will compete in quality and cost, which leads to improved and cheaper service. In such an open market, services set the price of their product.

We have chosen for an accounting mechanism where the service computes the cost of the operations it carries out on behalf of its clients. The reason for this is twofold. First, the provider of the service (*e.g.*, a human being, or an organisation) decides on the cost of the service: 'My files cost such-and-such per block, take it or leave it.' Second, the service is in the most practical position to carry out the computation of costs. This implies that clients have to trust services to 'keep their word,' they have to trust services to compute any costs according to the rules, published in advance.

Because, in an open system, clients can choose the service for carrying out their work, it can be expected that clients also choose a service they trust do the work properly and securely. A client, using file server X, trusts X not to reveal any files to other clients, for example. We therefore believe that it is acceptable that clients have to trust the services they use to some degree. If possible, however, clients should only have to have limited faith in the honesty of a service, that is, clients should not be forced to entrust their whole budget to a service; it would be too easy for a malicious user to set up a service to rob one or two clients of their budget. The accounting mechanisms must make it possible that clients entrust a limited amount to the service, an amount that covers the cost of the service exactly. A malicious service can still cheat in this set-up, but only by collecting the price of one unit of service without carrying it out. A client, robbed in this way, will never use any of the services offered by that user again, so the malicious user's gain will be negligible compared to the loss of trust of its clients.

Disk blocks cannot be compared to CPU seconds. Although the consumption of both is often expressed in the same unit, and sometimes must be expressed in money, we believe that the different nature of different resources and services requires a possibility to account for them in different ways, using different units and combinations of units. This allows much more freedom for resource management services to control access to different types of services independently.

7.1.4. Resource Control

Resource control mechanisms—or, perhaps, *service* control mechanisms—have the task of adjusting client's and service's budgets as services are provided, and of enforcing policies for fair distribution of resources and services in limited supply. These mechanisms should be general enough to be applicable to a wide class of services and resource management policies.

One area of resource control, *resource allocation*, is related to *scheduling*. Scheduling is the allocation of a resource to a number of competing processes, where the resource should be used optimally (maximise throughput) and each competitor receives an equal share of the resource (ensure fairness). Resource allocation mechanisms serve the same purpose, although often in a different time scale. CPU scheduling usually requires split-second decision making, while deciding about disk space allocation, in comparison, can be done at leisure.

We consider scheduling, which requires very rapid allocation decisions outside the scope of a resource control service. The delays alone, incurred by involving an extra service, are usually in excess of the time allowed for making scheduling decisions. In this chapter we concentrate on those services whose scheduling is possible on time scales that allow keeping records on file of past consumption and current allowances.

Records of service consumption must be kept in order to carry out any resource control and accounting. As stated in the previous section, a service provides the information about services consumed by its clients. In return, resource control mechanisms must provide services with information about clients' *'credit-worthiness'*—whether clients have a right for more service.

In principle, before carrying out a request, a service must check if the client is credit-worthy, that is, whether the client has any budget left or used up his fair share of service already. After carrying out the request, a service must register the consumption of service with a bookkeeping service. If the service-control service must be consulted before each request and informed after each request, network traffic will triplicate, and simple transactions will take more than three times longer. Clearly, this is unacceptable. Caching strategies are required with which services can avoid having to consult an accounting service for every transaction.

7.1.5. The Triangle Relation of Client, Service and Bank

When people go shopping, they make transactions: a service is rendered in return for money. These transactions between people and shops (clients and services) nearly always take place without need of a mutually trusted third party. One of the reasons for this is that shopkeepers trust that the money they receive is genuine. Money cannot easily be forged, so a person can only spend it once.

In transactions between computers, where the equivalent of money must be represented electronically, copying money is the easiest thing. Electronic transactions must therefore be handled differently. If money is represented by an (authenticated) bit pattern, nothing prevents a client from giving it to a number of different services in payment for services rendered (or to be rendered). Money would have no value as it could be copied without limit. Even sequence numbers on banknotes would not help: On each payment, the recipient would have to check whether the received money had already been spent somewhere else.

Instead of electronic money, electronic cheques could be used in payment. However, the use of these suffers from the same drawback as using electronic banknotes: the service must always check with the bank whether the cheque is covered; one cannot squeeze water out of stone. In The Netherlands, and possibly in many other countries as well, a guaranteed cheque exists. Banks have agreed to honour these cheques up to some maximum amount whether it is covered or not. By limiting the number of blank cheques in possession of an account holder, the banks limit their risk. Again, this system fails if clients can copy cheques at will: The risk to the banks will become too great.

The concepts of banknotes and of cheques can be combined to yield a better method of payment. A bank can hand out numbered banknotes payable to one service only. When a client presents such a banknote to the correct service, the service checks if it has not received a note with that number on it before; if this is indeed not the case, the note is genuine and it can be put in the bank at a later time. To prevent forgery, such notes could consist of the text *"This is banknote number thingumajig, with a value of thingumabob, intended for service what's-its-name"* signed by the bank server; that is, encrypted with the bank server's secret key.

This method does not require that a client or a server have to check with the bank on every transaction: a client can order a large number of banknotes at once, and a server can collect a large number of them before going to the bank. The method does require that the server keep a list of received banknote numbers; and, unless measures are taken, this list can become very long indeed. To restrict the size of the list, it is possible to require clients to hand over banknotes in order of increasing serial numbers. Two disadvantages remain, however, the least of which is that the amounts on the banknotes are fixed, so it is possible that quite a few banknotes must change hands to pay for one transaction; it is even possible that services must give *change*. The most important disadvantage is that the public key encryption required—to protect banknotes from forgery and to allow services to inspect the authenticity of received banknotes—makes banknotes inconveniently large (on the order of a thousand bytes) and therefore not very useful in applications.

In the next sections we shall examine a bank server structure that does not have these drawbacks. The basic idea is to have clients deposit a sufficient amount in the service's account before requesting it to do work. Naturally, the service must know which client deposits which amount, and later, when the client makes a request, it should be able to tell the request came from the same client that deposited the money. One method is to have each client pass some sort of identifying capability along with each request and each deposit, but the ***Amoeba*** distributed system has a signature mechanism that is ideal for just this purpose: When a client deposits money for a server at the bank, it provides a signature. The money received will then be labelled with that signature. When the client later makes a transaction with a service, it again includes its signature with the request. The service can then tell it is the same client that made the deposit.

7.2. Bank Service

As an example of a service for accounting and resource control, the Bank Service is described in this section. This service manages accounting information in a way that resembles the way a bank handles money. Different accounting units (*e.g.*, units representing disk blocks, cpu seconds) are represented by **virtual money** in different **currencies**, analogous to real money. Clients and services maintain **accounts** between which amounts of virtual money can be transferred.

7.2.1. Bank Accounts

The objects, managed by the Bank Service are **bank accounts**. There are two types, **private** and **business**. Both types of account can contain virtual money, and various requests are available to transfer virtual money from one account to another.

Bank accounts are protected by capabilities. There are rights that control withdrawal from an account, examination of an account, deposits into an account, etc. Services must know the identity of their clients in order to do accounting. Signatures* are used for client authentication.

Private accounts are much like regular bank accounts: After an account is created, virtual money can be deposited into it and withdrawn from it. Usually

* See § 2.3.2

a private account is manipulated solely by its owner. Business accounts are different. In contrast to private accounts, whose contents can be thought of as an amount of virtual money, a business account can be thought of as many amounts of virtual money, one for—or rather, from—each client that deposited virtual money into the account. With bank accounts in the real world, the holder is notified *who* deposits into it; with our business accounts this is not necessary, because deposited virtual money is automatically tagged with its source. The owner of a business account must specify a source when manipulating the account; otherwise, business accounts are little different from private accounts.

Business accounts are especially intended for services. Before a client can use a service that requires payment in the form of virtual money, a deposit must be made into the service's business account. Then requests can be sent to the service. The service, before carrying out a request, makes sure there is enough in its business account to pay for the request, then carries out the request, and finally transfers the cost of the request from the business account to a private account.

Although, in principle, the Bank Service is used as just described, it is rather inefficient when used exactly in this way. However, the design of the Bank Service allows some simple optimisations that make it quite efficient. When a client deposits into a service's business account, the amount will usually be sufficient to pay for a large number of requests—enough for a whole day's file service, for instance. The service, before honouring the first request, must examine the account, but once the amount in it is known, the service may carry out requests until that amount is consumed without further examining it; the client cannot secretly withdraw from the account behind the service's back, because the virtual money is already in the service's business account. The service can adjust its business account periodically; it is not necessary to make a transfer from the business account to a private account after each request.

Client crashes do not affect the Bank Service mechanisms at all. Server crashes do have consequences, if a server crashes after a request has been carried out, but before the business account has been adjusted. Server crashes are infrequent, however, and the damage is limited to the cost of a few transactions. The frequency with which services adjust their business accounts must be low to reduce network traffic, but an order of magnitude higher than the frequency of server crashes.

7.2.2. Capabilities and Signatures

Accounts are protected by capabilities. Authentication is done through the use of signatures. In this section we shall describe how capabilities and signatures are used for protection.

When a client creates a private account, two capabilities are returned, an *owner* capability with all rights on, and a *deposit* capability with only the right to deposit into the account. The client keeps the owner capability to himself, of course, but the deposit capability is usually given away to the service providing the virtual money (*e.g.*, a system manager).

When a business account is created, the Bank Server also returns two capabilities, an *owner* capability, and a *client* capability. The owner capability is kept carefully secret, but the client capability is published to the service's potential clients. This capability allows a client to deposit into the business account, and to read the balance of its own deposits into it. Once deposited, a client capability gives no right to withdraw from a business account.

Before explaining how *signatures* are used to authenticate clients, it is perhaps useful to quickly recall the signature mechanism presented in chapter 2, since it is used to provide authentication of requests sent to a service. A packet may contain a signature, a bit string that can be used as proof of the identity of the sender. A process, wishing to send a signed packet, puts his signature, S, in the signature field of the packet. Before the packet arrives, the *F-boxes* replace S by $F(X)$, where F is a one-way cipher. S can be viewed as a capability for producing $F(S)$. The mechanism is used as a means of proving 'This message comes from the same source as a previous one.'

Before depositing into a service's business account, a client chooses a signature, S, for identification. $F(S)$ is included in the request to the Bank Server to transfer an amount from its private account into the business account. This request contains two capabilities, a capability with a *withdraw* right for the client's private account and a capability with the *deposit* right for the service's business account; furthermore, the request contains $F(S)$ for identification in later requests to the service. The Bank Server, receiving this request, withdraws the amount from the client's account, and deposits it into the business account, tagged with $F(S)$.

Subsequently, the client sends requests to the service, each of which contains signature S in the signature field. Before the services receives the request, an F-box has converted S to $F(S)$. When a service receives the first request from a client, it sends a request to the Bank Server to read the balance of its business account tagged with $F(S)$. This request contains a capability for the business account with the *inspect* right and $F(S)$. If the amount read is sufficient, the request is carried out and an appropriate amount is transferred from the business account to a private account.

7.2.3. Maintainance of a Cache

Each service can maintain a cache, an entry consisting of a triple (s, b, c) of a client's signature, a balance, and the amount of service consumed. When a request is received, the cache is searched for an entry with the correct signature. If an entry is found, and the balance, b, is sufficient, the request is carried out, and the cost of the request is subtracted from b and added to c. If there is no entry for that client in the cache, a request is sent to the Bank Server to read the client's account. Then an entry is made with b set to the client's balance and c set to zero.

Periodically, the cache is examined, and for all cache entries with non-zero c, a request is sent to the Bank Server to transfer an amount c, tagged with s in its business account to a private account. The Bank Server returns the new balance (which is b or a larger amount, if the client deposited in the mean time) that the service uses to update the cache.

If, upon reception of a request, a service finds insufficient funds in its cache, it first updates the cache as described above to see if the client has made a deposit in the mean time. If the amount there is also insufficient, the request is not carried out, and a reply is sent in stead, requesting more funds.

7.2.4. Currencies

In a previous section the problem of comparing different kinds of service was broached. If only one accounting unit is used, it is difficult to exercise precise control over a number of different resources and services. What to do, for instance, if there is an ample supply of CPU cycles, but disk blocks are in scarce supply? Clearly, accounting mechanisms can be made more versatile if different units of accounting can be used for different kinds of resources.

Different units of accounting are represented by virtual money in different **currencies**. It is thus possible, for instance, to do accounting of disk blocks in *virtual guilders*, accounting of compilations in *virtual dinars*, and accounting of CPU seconds in *virtual yen*. New currencies can be created and removed dynamically, and conversion between currencies is possible. There is also the possibility of *minting* more virtual money in some currency, if that is useful.

Different currencies will be applied in different ways to obtain different types of accounting. This is illustrated by two simple examples. Suppose, the manager of a computer centre wants to set a maximum to the number of disk blocks that each user may occupy at any moment. This is implemented by creating a currency—say, *virtual roubles*—that represents disk blocks. When a user enters the system, he or she receives an amount of *virtual roubles*. When a disk block is allocated, an amount must be paid; when a block is released, the amount is returned. Suppose now that our manager wants to set a maximum number of CPU seconds per week that each user may consume. This is done by creating a currency—say, *virtual dollars*—that is used to pay each user a weekly salary to buy CPU seconds with. Several policies may be adapted for allowing users to take *virtual dollars* saved up into the next week, by making the salary depend on the amount left over at the end of the week.

These examples serve to illustrate that the provider of some services will wish to create a virtual currency to control the use of that service. For this reason, the Bank Service itself does not decide to create new currencies and the amounts to be minted, but the providers of the services in the system. Naturally, the Bank Server itself—being a provider of a service also—may create some currency that controls the creation of other currencies, for instance, to prevent any one user from creating too many currencies.

Services can ask to be paid in combinations of currencies. A phototypesetter service, for instance, may adopt the policy to demand payment in real money (represented by some virtual currency) to cover the cost of photographic material (paper, developer), in addition to payment in another virtual currency that prevents a (rich) user from occupying the phototypesetter too long at a time.

7.2.5. Bank Server Requests

The commands to the Bank Server can be divided into two categories: those that manage accounts, and those that manage currencies. Not all requests will be discussed here, but a subset that amply illustrates how Bank Service is used. Before the Bank Service requests are described in detail, however, a few remarks on notation are necessary. Some requests, such as the request to transfer an amount from one account to another, operate on two objects. In these cases, one object (*e.g.*, the account from which the withdrawal is made) will be the **primary** object, that is, the object indicated by the capability of the request, while the second object will be the **secondary** object, that is, the object whose capability is contained in the request's data. In the notation used, the first parameter of each request is special; it indicates the *capability* field of the request. Although the Bank Service does require signatures in some of its requests, the *signature* field is not needed in requests to the Bank Service; only *public signatures* have to be sent to it. Note that this allows Bank Service to use its own accounting mechanisms on its clients. When a capability refers to an object, and no confusion results, the same names will be used for the capability for the object and the name of the object.

CreateBusinessAccount(*BankService*);
returns: *OwnerCapability*, *DepositCapability*

> The *BankService* capability in this request is the public capability that allows all clients to the Bank Service to create new objects. The Bank Service will create an empty business account and return two capabilities for it, an owner capability and a capability to give to clients to deposit into the account.

CreatePrivateAccount(*BankService*);
returns: *OwnerCapability*, *DepositCapability*

> This request creates an empty private account and, like the *CreateBusinessAccount* request, returns two capabilities for it.

Transfer(*FromAccount*, *FromSignature*, *ToAccount*, *ToSignature*, *Amount*);
returns: Balance of *FromAccount*

> This request is used to transfer amounts from one account (business or private) to another (business or private). The signature parameters are only needed for transfers from and to business accounts; when referring

to a private account, the signatures can be dummy parameters. Note, that the signatures need not be in the request's *signature* field, the public signature suffices. *Amount* represents a list of one or more virtual currencies and amounts.

Inspect(*Account*, *signature*);
returns: Balance of *Account*

The Bank Server inspects the account indicated by *Account*, and returns its balance. If it is a business account, *signature* can be used to tell the Bank Server to consider only the balance of the account, deposited under that signature.

RemoveAccount(*Account*);
returns: Acknowledgement

The Bank Server removes the account (business or private) after making sure it is empty. If this is not the case, the account is not removed, and an error reply is generated.

CreateCurrency(*BankServer*, *Account*, *Amount*);
returns: *MintCapability*, *Name*

A new currency is generated, of which *Amount* is minted and put in *Account*. A capability for controlling the currency is returned, plus a small integer which functions as the name of the new currency.

Mint(*MintCapability*, *Account*, *Amount*);
returns: Acknowledgement

Amount is minted of the currency to which *MintCapability* refers; it is deposited into *Account*.

RemoveCurrency(*MintCapability*);
returns: Acknowledgement

A currency is removed. All amounts of the currency in all accounts are deleted.

7.3. Accounting Policies

The Bank Service mechanisms, presented in the previous section, can only be useful if they serve an efficient implementation of a wide variety of accounting policies. In this section some of the existing accounting policies will be reviewed, and it will be shown how they can be realised.

7.3.1. Payment for Services

An important application of accounting is in computing the cost of services consumed. This is not only useful in commercial computer centres that must bill their clients in accordance with their service consumption, but also for educational institutes, where it is useful, for example, to charge students for excessive use of expensive resources (*e.g.*, line printer paper, phototypesetter pages).

Accounting for cost of services is one of the simplest applications of Bank Service. In principle, each user (periodically) receives an amount in some currency, representing a *budget* or *salary*. This currency is used to pay for services consumed, as described earlier. At any time, the difference between the sum given to the user and the amount left over in the user's account represents the amount consumed.

For accounting purposes, a service is considered to *let* its resources to clients and carry out operations for clients. In principle, charges can be levied on the operations, and hire on the consumption of resources. The charge on execution of requests is a function of the cost of carrying them out, the hire of resources is a function of the type, amount and time the resource is consumed.

A file server, for instance, stores files and carries out operations on them. It can charge an amount for each operation carried out, plus an additional sum per day for the consumed file space. Several policies can be adopted when clients have insufficient funds to pay for file service. It can start to refuse to carry out operations for its clients, it can also make client files inaccessible until sufficient budget is once again available, and, in extreme cases, it can altogether remove unpaid for client files.

The charges levied on execution of requests are collected after each request. Charges on resource hire can be collected in several ways. They can be collected when the resource is returned, but this may result in clients keeping a resource past the point where their budget runs out. The charges can be collected periodically—say, every hour, or minute, depending on the resource. But they can also be collected whenever a client requests an operation on the resource; charges are collected then, anyway.

7.3.2. Quota, Budgets and Salaries

To prevent a user from using too much of a resource at once, *quota* are usually established. A **quotum** is a maximum amount that a user may consume of a resource. Examples of quota are n disk blocks, m magnetic tapes, p CPU seconds per day, q line printer pages per week. Quota can be separated into two kinds: *absolute* and *per unit of time.* Absolute quota are used for *reallocable* resources, resources that a client uses for some time, and then gives back so they can be used by another. Quota per unit of time are used for *allocate-once* resources, resources that a client uses up and can not be returned to be used by another. Disk blocks and mag tapes are examples of reallocable resources, CPU seconds and phototypesetter pages are examples of allocate-once resources.

Quota for reallocable resources are simple to implement. Every user, when given access to the system receives a **budget**, amounts of virtual currency representing maxima on resources that may be used. Whenever the user requests more resources, they must be paid for from the budget. Whenever the user returns resources, the payment is returned.

When more of a resource becomes available, or fewer users have to share the resource, budgets may be increased; when more users have to share a resource, or the resource becomes scarcer, budgets must be decreased. It is a simple matter to increase users' budgets, but it is harder to decrease them: the complete budget may be invested, so there may not be enough budget left to take away. An approach to accomplish decreasing budgets can be to *devaluate* the currency representing the resource, that is, make the resource more expensive. However, this will favour users that obtained their resources at the old price, and they will not be willing to return any of their resources, because they will not be able to obtain the same amount later. In most cases, it is not possible to take away resources from users by pure mechanical means, anyway: when more users have to share the same disk space, for instance, the old users must make room for the new; however, the old users will have valuable information stored in their disk space, and the 'system' cannot just take away some of the old users' disk blocks.

For allocate-once resources a slightly different approach can be used, based on **salaries**. Every user receives an hourly, daily, monthly (or whatever is appropriate) salary with which to pay for consumption of allocate-once resources. The salary can be independent of the amount left over in the users' accounts, which makes it possible for users to save up for future use. The salary can also be made dependent of the left-over amount, to restrict saving, or to prevent it altogether.

Often, it will be useful to combine budgets and salaries. For instance, a user could be given a budget to restrict the number of tape drives used simultaneously, and a salary to restrict the time the tape drives are allocated.

8

SAMENVATTING

Dit proefschrift beschrijft principes voor het ontwerp van gespreide besturingssystemen. Met behulp van deze principes is het de bedoeling gebruik te maken van de mogelijkheden die gespreide systemen kunnen bieden: In een gespreid systeem zijn meerdere processoren zodanig opgesteld dat ze informatie kunnen uitwisselen door middel van een computernetwerk, maar dat bij een storing in een willekeurige processor de andere processoren kunnen blijven functioneren.

Een van de belangrijkste doelstellingen van het hier beschreven onderzoek is om van deze eigenschap gebruik te maken door een besturingssysteem te ontwerpen dat, indien een willekeurige component uitvalt (apparatuur of programmatuur), de gebruikersprogrammatuur zo veel mogelijk ongestoord blijft doorwerken, iets langzamer wellicht. De technieken om deze doelstelling te bereiken zijn het ontwerpen van fout-tolerante programmatuur—programmatuur waarin rekening wordt gehouden met de mogelijkheid dat er fouten optreden—, en replicatie van apparatuur, programmatuur en gegevens.

Een andere belangrijke doelstelling is te verwezenlijken dat de capaciteit van een gespreid systeem de som van de capaciteiten van zijn onderdelen zoveel mogelijk benadert; de extra betrouwbaarheid van een gespreid systeem moet zo min mogelijk kosten. Dit betekent dat in het systeem een hoge graad van parallellisme moet worden nagestreefd en een derde belangrijke doelstelling is de consequenties van dit parallellisme te beheersen, zodat, wanneer bijvoorbeeld twee gebruikers van het systeem gelijktijdig eenzelfde bestand veranderen, er geen onverwachte gevolgen optreden.

Hoofdstuk 1 is een inleiding in de problematiek van gespreide systemen en bevat een overzicht van relevant onderzoekswerk elders.

In Hoofdstuk 2 worden de interproces communicatie mechanismen beschreven. Verschillende methoden om communicerende partijen te benoemen worden opgesomd en het mechanisme wordt beschreven om communicerende processen elkanders identiteit te laten vaststellen. Met behulp van dit mechanisme ontstaat de mogelijkheid onbevoegde processen de toegang tot bepaalde *services* onmogelijk te maken. Dit mechanismne maakt gebruik van zogenaamde *F-boxes*, die zowel in *hardware* als *software* kunnen worden gerealiseerd. Het protectiemechanisme maakt gebruik van de aanname dat tussen elk tweetal communicerende processen een F-box aanwezig is die niet omzeild kan worden. Hiertoe kunnen software F-boxes in de *operating system kernel* worden opgenomen, of in een *switching node* van het netwerk; hardware F-boxes kunnen op de *network interfaces* worden aangebracht.

Het slot van Hoofdstuk 2 beschrijft methoden om *objecten* in een computer netwerk te localiseren. Voor een bepaalde klasse van *locate* algoritmen wordt een ondergrens voor het aantal boodschappen bewezen.

Hoofdstuk 3 gaat over het beheer van *objecten* door *services*. Als een *client* proces een operatie op een object (bijv. een *file*) wil uitvoeren, dan wordt een *request* naar de *service* gestuurd die het object beheert. Een van de *server* processen voor de betreffende service voert het verzoek uit en retourneert een *reply*. Om het recht aan te tonen om een operatie op een object uit te voeren moet de client een *capability* overleggen, een soort toegangskaartje voor een object. Deze capability wordt tevens gebruikt als middel het object te adresseren. Het beheer van capabilities in een gespreid systeem komt uitgebreid aan de orde. Methoden worden beschreven om operaties op capabilities uit te voeren en om ze op een practische, doch veilige manier te kunnen bewaren.

Een ander onderwerp dat in Hoofdstuk 3 behandeld wordt is het mechanisme om op snelle, betrouwbare en veilige manier *requests* en *replies* tussen clients en servers over te brengen. Dit mechanisme is geïmplementeerd en blijkt een bijzonder efficient interprocescommunicatiemechanisme op te leveren.

Het beheer van processen komt in Hoofdstuk 4 aan de orde. Processen kunnen worden opgesplitst in *tasks*, die vrij onafhankelijk van elkaar in *shared memory* draaien. Een proces wordt geladen, verhuisd, *checkpointed* of *debugged* met behulp van de *Process Descriptor*, waarmee de toestand van een proces wordt beschreven.

Een aantal services worden benut bij het beheer van processen. De *Process Server* beheert een *Processor Pool* waaruit processoren worden toegwezen om een programma te draaien. De *Loader Service* wordt gebruikt voor opslag van *binaries* en de *Bootstrap Service* detecteert storingen en repareert services die er de dupe van worden.

In Hoofdstuk 5 wordt een techniek beschreven voor process *scheduling*, waarbij de looptijd tot dusver gebruikt wordt om een schatting te maken van de resterende looptijd. De schattingsfunctie die in dit hoofdstuk wordt afgeleid is gebaseerd op metingen aan proces looptijden in het Unix systeem. Het gedrag van de ontwikkelde algoritme wordt geïllustreerd door middel van simulatie.

File Service is het onderwerp van Hoofdstuk 6. De hier beschreven Amoeba File Server heeft een gelaagde structuur en realiseert mechanismen voor replicatie, *concurrency control* en *version management*. De Amoeba File Server wordt gekarakteriseerd door zijn toepasbaarheid op optische disks, door het gebruik van *optimistic concurrency control* en door de manier waarop op alle niveaus efficient *caches* kunnen worden bijgehouden.

Centraal in Hoofdstuk 7 staat de *Bank Service* die dient om clients en services in staat te stellen *accounting* en *resource control* te doen. Traditionele accounting mechanismen registreren slechts het gebruik van processor, (achtergrond)geheugen en I/O; de Bank Server is ontworpen om alle gebruikers van een systeem in staat te stellen services aan te bieden in ruil voor 'betaling,' hetzij in de vorm van geld, hetzij in de vorm van tegen-services.

Hoofdstuk 8 bevat een nederlandstalige samenvatting en Hoofdstuk 9 een bibiografie.

9

REFERENCES

1. Abramowitz, M., *Handbook of Mathematical Functions with Formulas Graphs and Mathematical Tables,* Applied Mathematics Series (NBS); 55 (1972).
2. Akkoyunlu, R., Bernstein, A., and Schantz, R., 'Interprocess Communication Facilities for Network Operating Systems,' Technical Report No. 25, Dept. of Computer Science, State University of New York, Stony Brook (March 1972).
3. Ball, J. E., Burke, E. J., Gertner, I., Lantz, K. A., and Rashid, R. F., 'Perspectives on Message-Based Distributed Computing,' *Proc. IEEE*, (1979).
4. Berlekamp, E. R., 'Factoring polynomials over large finite fields,' *Mathematics of Computation 24*(111) pp. 713-735 (July 1970).
5. Boggs, D. R., Shoch, J. F., Taft, E. A., and Metcalfe, R. M., 'Pup: An Internetwork Architecture,' *IEEE Trans. Comm. Com-28*(4) pp. 612-623 (April 1980).
6. Brinch-Hansen, P., 'The Nucleus of a Multiprogramming System,' *Comm. ACM 13*(4) pp. 238-250 (April 1970).
7. Brown, M. R., Kolling, K., and Taft, E. A., 'The Alpine File System,' *to appear in ACM TOCS*, (1985).
8. Cheriton, D. R., Malcolm, M. A., Melen, L. S., and Sager, G. R., 'Thoth, a Portable Real-Time Operating System,' *Comm. ACM 22*(2) pp. 105-115 (February 1979).
9. Cheriton, D. R. and Zwaenpoel, W., 'The Distributed V Kernel and its Performance for Diskless Workstations,' *Operating Systems Review 17*(5) pp. 129-140 (October 1983).

10. Coffman, E. G. Jr. and Denning, P. J., *Operating Systems Theory,* Prentice-Hall, Inc., Englewood Cliffs, N.J. (1973).

11. Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R., *Structured Programming,* Academic Press, New York (1972).

12. Dalal, Y. K. and Metcalfe, R. M., 'Reverse Path Forwarding of Broadcast Packets,' *Comm. ACM 21*(12) pp. 1040-1048 (December 1978).

13. Dalal, Y. K., 'Broadcast Protocols in Packet Switched Computer Networks,' Stanford Electronics Lab Tech. Report 128, Stanford University (April 1977).

14. Denning, P. J., 'The Working Set Model for Program Behavior,' *Comm. ACM 11*(5) pp. 323-333 (May 1968).

15. Dennis, J. B. and Horn, E. C. van, 'Programming Semantics for Multiprogrammed Computations,' *Comm. ACM 9*(3) pp. 143-155 (March 1966).

16. Diffie, W. and Hellman, M. E., 'Multiuser Cryptographic Techniques,' *Proc. NCC,* pp. 109-112 (1967).

17. Dijkstra, E. W., 'Self-Stabilizing Systems in Spite of Distributed Control,' *Comm. ACM 17*(11) pp. 643-644 (November 1974).

18. Dion, J., 'The Cambridge File Server,' *Operating System Review 14*(4) pp. 26-35 (Oct. 1980).

19. Donnelley, J. E., 'Managing Domains in a Network Operating System,' *Proc. On-Line Conf. on Local Networks & Distributed Office Systems,* pp. 345-361 (May 1981).

20. Ellis, C. A., 'Consistency and Correctness of Duplicate Database Systems,' *Proc. Sixth Symp. on Oper. Syst. Prin.,* (November 1977).

21. Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., 'The Notions of Consistency and Predicate Locks in a Database Operating System,' *Comm. ACM 19*(11) pp. 624-633 (November 1976).

22. Evans, A., Kantrowitz, W., and Weiss, E., 'A User Authentication Scheme Not Requiring Secrecy in the Computer,' *Comm. ACM 17*(8) pp. 437-442 (August 1974).

23. Farber, D. J. and Larson, K. C., 'The System Architecture of The Distributed Computer System—The Communication System,' *Polytechnic Institute of Brooklyn Symp. on Computer Networks,* (April 1972).

24. Fridrich, M. and Older, W., 'The Felix File Server,' *Proc. Eighth Symp. on Oper. Syst. Prin.,* pp. 37-44 (December 1981).

25. Gelernter, D. and Bernstein, A. J., 'Distributed Communication via a Global Buffer,' *Proc. 1st ACM SIGACT-SIGOPS Symp. on Princ. of Distributed Computing,* pp. 10-18 (1982).

26. Gonzalez, M. J., Jr., 'Deterministic Processor Scheduling,' *Computing Surveys 9*(3) pp. 173-204 (September 1977).

27. Hofstadter, D. R., *Gödel, Escher, Bach: An Eternal Golden Braid,* Hassocks: Harvester (1979).

28. Jones, A. K. and Wulf, W. A., 'Towards the Design of Secure Systems,' *Software Practice & Experience 5* pp. 321-326 (1975).

29. Jones, A.K., 'The Object Model--A Conceptual Tool for Structuring Software,' in *Operating Systems - An Advanced Course*, ed. R. Bayer,R.M. Graham andG. Seegmuller, Springer Verlag (1978).

30. Jones, A. K. and Liskov, B. H., 'A Language Extension for Expressing Constraint on Data Access,' *Comm. ACM 21*(5) pp. 358-367 (May 1978).

31. Jr., T. B. Steel, 'UNCOL: The Myth and the Fact,' pp. 325-344 in *Ann. Rev. Auto. Prog.*, ed. R. Goodman, (1960).

32. Knuth, D. E., *The Art of Computer Programming, Vol. II,* Addison Wesley, Reading, Mass (1969).

33. Kung, H. T. and Robinson, J. T., 'On Optimistic Methods for Concurrency Control,' *ACM Transactions on Database Systems 6*(2) pp. 213-226 (June 1981).

34. Lampson, B. W., 'A Note on the Confinement Problem,' *Comm. ACM 6*(10) pp. 613-615 (October 1973).

35. Lampson, B. W. and Sturgis, H., *Crash Recovery in a Distributed Storage System,* Xerox PARC, Palo Alto, CA. (1979).

36. Lampson, B. W. and Sproull, R. F., 'An Open Operating System For A Single User Machine,' *Proc. Seventh Symp. on Oper. Syst. Prin.*, pp. 98-105 (1979).

37. Lampson, B. W., 'Hints for Computer System Design,' *Proc. 9th SOSP,* (Oktober 1983).

38. Lann, G. Le, 'Algorithms for Distributed Data-Sharing Systems which use Tickets,' *Third Berkeley Workshop on Distributed data Management and Computer Networks,* pp. 259-272 (August 1978).

39. LeVeque, W. J., *Topics in Number Theory,* Addison Wesley, Reading, Mass. (1956).

40. Liskov, B. and Zilles, S., 'Programming With Abstract Data Types,' *SIGPLAN Notices 9* pp. 50-59 (April 1974).

41. McQuillan, J. M., 'Adaptive Routing Algorithms for Distributed Computer Networks,' Ph.D. Thesis, Div. of Engineering and Applied Sciences, Harvard University (1974).

42. Metcalfe, R. M. and Boggs, D. R., 'Ethernet: Distributed Packet Switching for Local Computer Networks,' *Comm. ACM 19*(7) pp. 395-404 (July 1976).

43. Miller, L. W. and Schrage, L. E., 'The Queue M/G/1 with the Shortest Remaining Processing Time Discipline,' *Operations Research 14*(4) pp. 670-683 (July-August 1966).

44. Millstein, R. E., 'The National Software Works: A Distributed Processing System,' *Proc. ACM Annual Conf.*, pp. 44-52 (1977).

45. Mueller, E.T., Moore, J.D., and Popek, G.J., 'A nested transaction mechanism for LOCUS,' *Proc. Ninth Symp. on Oper. Syst. Prin.*, pp. 71-90 (1983).

46. Mullender, S.J. and Tanenbaum, A.S., 'A Distributed File Server Based on Optimistic Concurrency Control,' *Proc. Tenth Symp. on Oper. Syst. Prin.*, (December 1985).

47. Mullender, S. J. and Renesse, R. van, 'A Secure High-Speed Transaction Protocol,' *Proceedings of the Cambridge EUUG Conference*, (1984).

48. Mullender, S. J. and Tanenbaum, A. S., 'Protection and Resource Control in Distributed Operating Systems,' *Computer Networks 8*(5,6) pp. 421-432 (1984).

49. Mullender, S. J., *Principles of Distributed Operating System Design,* PhD. Thesis, CWI, Amsterdam (October 1985).

50. Needham, R. M. and Herbert, A. J., *The Cambridge Distributed Computer System,* Addison-Wesley, Reading, Mass. (1982).

51. Nelson, B. J., 'Remote Procedure Call,' CSL-81-9, Xerox PARC (1981).

52. Purdy, G. B., 'A High Security Log-in Procedure,' *Comm. ACM 17*(8) pp. 442-445 (August 1974).

53. Rashid, R. and Robertson, G., 'Accent: A Communication Oriented Network Operating System Kernel,' *Proc. Eighth Symp. on Oper. Syst. Prin.*, pp. 64-75 (December 1981).

54. Reed, D., 'Naming and Synchronization in a Decentralized Computer System,' *PhD. Thesis*, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, (1978).

55. Reed, D. and Svobodova, L., 'SWALLOW: A Distributed Data Storage System for a Local Network,' *Proc. IFIP*, pp. 355-373 (1981).

56. Reed, D. P., 'Synchronization with Eventcounts and Sequencers,' *Sixth ACM Symposium on Operating System Principles*, (November 1977).

57. Rihaczek, K., 'OSIS Open Shop for Information Services,' *to be published in IFIP Computer Compacts*, North-Holland, (1985).

58. Rivest, R. L., Shamir, A., and Adleman, L., 'A Method for Obtaining Digital Signatures and Public Key Cryptosystems,' *Comm. ACM 21*(2) pp. 120-126 (February 1978).

59. Robinson, J. T., 'Design of Concurrency Controls for Transaction Processing Systems,' Ph.D Thesis (CMU-CS-82-114), Carnegie-Mellon University, Pittsburgh Pa. (April 1982).

60. Rochkind, M.J., 'The Source Code Control System,' *IEEE Trans. on Softw. Eng. SE-1*(4) pp. 364-370 (Dec. 1975).

61. Satyanarayanan, M., 'The ITC Distributed File System: Principles and Design,' *Proc. Tenth Symp. on Oper. Syst. Prin.*, (December 1985).

62. Schlageter, G., 'Optimistic Methods for Concurrency Control in Distributed Database Systems,' *Proc. VLDB Conference*, (1981).

63. Schroeder, M. D., Gifford, D. K., and Needham, R. M., 'A Caching File System for a Programmer's Workstation,' *Proc. Tenth Symp. on Oper. Syst. Prin.*, (December 1985).

64. Shamir, A., 'How to Share a Secret,' *Comm. ACM 22*(11) pp. 612-613 (November 1979).

65. Shoch, J. F., 'Internetwork Naming, Addressing, and Routing,' *Proc. 17th IEEE Comput. Soc. Int. Conf. (CompCon)*, (Sept 1978).

66. Sloane, N. J. A., *A Handbook of Integer Sequences,* Academic Press, New York (1973).

67. Solomon, M. H. and Finkel, R. A., *The Roscoe Distributed Operating System.* 1979.

68. Spector, A. Z., 'Performing Remote Operations Efficiently on a Local Computer Network,' *Comm. ACM 25*(4) pp. 246-260 (April 1982).

69. Standards, National Bureau of, 'Data Encryption Standard,' *Fed. Inf. Process. Stand. Publ. 46*, (January 1977).

70. Stonebraker, M., 'Operating System Support for Database Management,' *Comm. ACM 24*(7) pp. 412-418 (July 1981).

71. Sturgis, H., Mitchell, J.G., and Israel, J., 'Issues in the Design and Use of a Distributed File System,' *Operating System Review 14*(3) pp. 55-69 (July 1980).

72. Tanenbaum, A. S., *Computer Networks,* Prentice-Hall, Englewood Cliffs, N.J. (1981).

73. Tanenbaum, A. S. and Mullender, S. J., 'Operating System Requirements for Distributed Data Base Systems,' pp. 105-114 in *Distributed Data Bases*, ed. H. J. Schneider, North-Holland Publishing Co. (1982).

74. Tanenbaum, A. S., Staveren, J. M. van, Keizer, E. G., and Stevenson, J. W., 'A Practical Tool Kit For Making Portable Compilers,' *Comm. ACM 26*(9) pp. 654-660 (September 1983).

75. Thomas, R. H., 'A Resource Sharing Executive for the ARPANET,' *Proc. NCC*, pp. 155-163 (1973).

76. Thomas, R. H., *A Solution to the Update Problem for Multiple Copy Databases which uses Distributed Control,* BBN, Cambridge, Ma. (1976).

77. Walker, B., Popek, G. J., English, R., Kline, C., and Thiel, G., 'The LOCUS Distributed Operating System,' *Proc. Ninth Symp. on Oper. Syst. Prin.*,

78. Ward, S. A., 'TRIX: A Network-Oriented Operating System,' *proc. IEEE*, pp. 344-349 (1980).

79. Watson, R. W. and Fletcher, J. G., 'An architecture for Support of Network Operating System Services,' *Computer Networks 4*(1) pp. 33-49 (February 1980).

80. Wilkes, M. V., *Time-Sharing Computer Systems,* American Elsevier, New York (1968).

81. Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F., 'HYDRA: The Kernel of a Multiprocessor Operating System,' *Comm. ACM 17*(6) pp. 337-345 (June 1974).

VRIJE UNIVERSITEIT TE AMSTERDAM

PRINCIPLES OF DISTRIBUTED OPERATING SYSTEM DESIGN

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor in
de wiskunde en natuurwetenschappen aan
de Vrije Universiteit te Amsterdam,
op gezag van de rector magnificus
dr. P. J. D. Drenth,
hoogleraar in de faculteit der sociale wetenschappen,
in het openbaar te verdedigen
op donderdag 31 oktober 1985 te 13.30 uur
in het hoofdgebouw der universiteit, De Boelelaan 1105

door

SAPE JURRIËN MULLENDER

geboren te Nieuwer Amstel

1985

MATHEMATISCH CENTRUM, AMSTERDAM

Promotor : prof. dr. A. S. Tanenbaum
Referent : dr. P. J. Weinberger

STELLINGEN

1. In gespreide systemen kan voor betrouwbare protectiemechanismen i.h.a. niet vertrouwd worden op de *operating system kernel*.

   S. J. Mullender en A. S. Tanenbaum, "Protection and Resource Control
in Distributed Operating Systems," *Computer Networks*, 8(5,6) (1984)

2. De operating system services van Amoeba kunnen—ondersteund door geschikte services—een goede basis vormen voor database toepassingen.

   M. Stonebraker, "Operating System Support for Database
   Management," *Comm. ACM* 24(7), pp 412-418 (1981)

3. In zeer grote gespreide systemen is het belangrijk dat er van *locality of reference* gebruik gemaakt kan worden. Hiertoe moet bij allocatie van *resources* onderscheid gemaakt kunnen worden tussen *"local"* en *"remote" resources*.

4. In locale netwerken is de *scarce resource* meestal CPU bandbreedte en zelden netwerk bandbreedte.

   S.J. Mullender en R. van Renesse, "A Secure High-Speed Transaction
   Protocol," *Proc. EUUG Conf*, Cambridge, UK (1984)

5. Bij het ontwerpen van efficiente interprocescommunicatiemechanismen dient gebruik te worden gemaakt van *information hiding*; bij gespreide systemen betekent dit dat de communicatie*interface* welgedefinieerd moet zijn, maar dat de daarvoor gebruikte communicatie*protocollen* heel goed netwerkafhankelijk kunnen zijn.

   D. L. Parnas, "On The Criteria To Be Used In Decomposing Systems
   Into Modules," *Comm. ACM*, 15(12), pp. 1053-1058 (1972)

   S.J. Mullender en R. van Renesse, "A Secure High-Speed Transaction
   Protocol," *Proc. EUUG Conf*, Cambridge, UK (1984)

6. "Parallel programming" moet aan machines worden overgelaten; mensen hebben hiervoor geen aanleg.

7. Indien dezelfde interprocescommunicatieprotocollen gebruikt worden voor zowel *local-area networks* als *wide-area networks* verdient het aanbeveling ze te optimaliseren voor locale communicatie.

   "Distributed Systems Management in Wide-Area Networks,"
   (ed. A. Langsford), *COST-11 bis DSM report*, publ. by AERE Harwell

   B. W. Lampson, "Hints for Computer System Design," *Proc. Ninth
   Symp. on Oper. Syst. Prin.*, pp. 33-48 (1983)

8. Bij het ontwerpen van file systemen dient men zich te realiseren dat de meeste files klein zijn.

S.J. Mullender en A.S. Tanenbaum, *Immediate Files*, Software P&E 14,4 (1984)

9. Het geven van programmeeronderricht op middelbare scholen omdat "iedereen tegenwoordig met computers te maken heeft," is vergelijkbaar met het onderwijzen van autotechniek omdat "iedereen tegenwoordig auto rijdt."

10. Een diploma geldt als een bewijs van deskundigheid. In Nederland blijkt ook het omgekeerde te gelden: Geen diploma geldt als een bewijs van ondeskundigheid. (Voor politici bestaat overigens geen diploma.)