CWI Tracts

Managing Editors

J.W. de Bakker (CWI, Amsterdam)
M. Hazewinkel (CWI, Amsterdam)
J.K. Lenstra (CWI, Amsterdam)

Editorial Board

W. Albers (Enschede)
P.C. Baayen (Amsterdam)
R.J. Boute (Nijmegen)
E.M. de Jager (Amsterdam)
M.A. Kaashoek (Amsterdam)
M.S. Keane (Delft)
J.P.C. Kleijnen (Tilburg)
H. Kwakernaak (Enschede)
J. van Leeuwen (Utrecht)
P.W.H. Lemmens (Utrecht)
M. van der Put (Groningen)
M. Rem (Eindhoven)
A.H.G. Rinnooy Kan (Rotterdam)
M.N. Spijker (Leiden)

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The CWI is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O).

CWI Tract 41

The Amoeba distributed operating system: Selected papers 1984 - 1987

edited by
Sape J. Mullender



Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

1980 Mathematics Subject Classification: 68A05.
1987 CR Categories: C0, C2, D4.
ISBN 90 6196 325 7
NUGI-code: 811

Copyright © 1987, Stichting Mathematisch Centrum, Amsterdam
Printed in the Netherlands

Preface

This tract contains selected articles relating to the Amoeba Distributed Operating System which were published between 1984 and 1987. The papers reflect a joint effort between the Centre for Mathematics and Computer Science, and the Vrije Universiteit, both located in Amsterdam, the Netherlands. Any citations should refer to the original publications rather than this collection.

Contents

INTRODUCTION 1

Distributed Operating Systems 3
A. S. TANENBAUM and R. VAN RENESSE
ACM Computing Surveys
Vol. 17, No. 4, pp. 419-470
December 1985

The Design of a Capability-Based Distributed Operating System 77
S. J. MULLENDER and A. S. TANENBAUM
The Computer Journal
Vol. 29, No. 4, pp. 289-300
March 1986

PROTECTION 101

Using Sparse Capabilities in a Distributed Operating System 103
A. S. TANENBAUM, S. J. MULLENDER, and R. VAN RENESSE
Proc. 6th Int. Conf. on Distributed Computing Systems
pp. 558-563
May 1986

Capability-Based Protection in Distributed Operating Systems 115
A. S. TANENBAUM, R. VAN RENESSE, and S. J. MULLENDER
Proceedings of Symposium Certificering van Software
Utrecht, Netherlands
November 1984

PROTOCOLS 123

A Secure High-Speed Transaction Protocol 125
S. J. MULLENDER and R. VAN RENESSE
Proceedings of the Cambridge EUUG Conference
September 1984

Distributed Match-Making for Processes in Computer Networks 137
S. J. MULLENDER and P. M. B. VITANYI
Proceedings 4th ACM Principles of Distributed Computing
Minaki, Canada
August 1985

RELIABILITY 163

Reliability Issues in Distributed Operating Systems 165
A. S. TANENBAUM and R. VAN RENESSE
Proc. 6th Symp. Reliability of Distr. Softw. & Datab. Syst.
Williamsburg, Virginia
pp. 3-11
March 1987

FILE SYSTEM 183

A Distributed File Service Based on Optimistic Concurrency Control 185
S. J. MULLENDER and A. S. TANENBAUM
Proceedings of the 10th Symposium on Operating Systems Principles
Orcas Island, Washington.
pp. 51-62
December 1985

Immediate Files 209
S. J. MULLENDER and A. S. TANENBAUM
Software-Practice and Experience
Vol. 14, No. 4, pp. 365-368
April 1984

WIDE-AREA NETWORKS 215

Distributed Systems Management in Wide-Area Networks 217
S. J. MULLENDER
Proc. NGI/SION Symposium
Amsterdam
pp. 415-424
April 1984

Connecting RPC-Based Distributed Systems Using Wide-Area Networks 231
R. VAN RENESSE, A. S. TANENBAUM, J. M. VAN STAVEREN, and J. HALL
Informatica Report IR-118
Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam
December 1986

APPLICATIONS 245

A Distributed, Parallel, Fault Tolerant Computing System 247
H.E. BAL, R. VAN RENESSE, and A.S. TANENBAUM
Informatica Report IR-106
Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam
October 1985

Parallel and Distributed Compilations in Loosely-Coupled Systems 261
ERIK H. BAALBERGEN
Proc. of workshop "Large Grain Parallelism"
Providence, RI
Oct. 1986

Parallel Alpha-Beta Search 267
H.E. BAL and R. VAN RENESSE
Proc. NGI-SION Symposium Stimulerende Informatica
pp. 379-385
Utrecht, Netherlands
April 1986

EXPERIENCE 279

Making Distributed Systems Palatable 281
ANDREW S. TANENBAUM and ROBBERT VAN RENESSE
Position Paper in 2nd SIGOPS workshop "Making Distributed Systems Work"
Amsterdam, Netherlands
September 1986

Making Amoeba Work 285
SAPE J. MULLENDER
Position Paper in 2nd SIGOPS workshop "Making Distributed Systems Work"
Amsterdam, Netherlands
September 1986

From UNIX to a Usable Distributed Operating System 289
R. VAN RENESSE
Proceedings of the EUUG Autumn '86 Conference
Manchester, UK
pp. 15-21
September 1986

Accommodating Heterogeneity in the Amoeba Distributed System 297
SAPE J. MULLENDER and ROBBERT VAN RENESSE
Proceedings of SOSP Heterogeneity Workshop
Orcas Island, Washington, USA
December 1985

Connecting UNIX Systems Using a Token Ring 301
ROBBERT VAN RENESSE, ANDREW S. TANENBAUM, and SAPE J. MULLENDER
Proceedings of the Cambridge EUUG Conference
September 1984

CONTRIBUTING AUTHORS 309

Introduction

Distributed Operating Systems

Andrew S. Tanenbaum
Robbert van Renesse
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Distributed operating systems have many aspects in common with centralized ones, but they also differ in certain ways. This paper is intended as an introduction to distributed operating systems, and especially to current university research about them. After a discussion of what constitutes a distributed operating system, and how it is distinguished from a computer network, various key design issues are discussed. Then several examples of current research projects will be examined in some detail, namely the Cambridge Distributed Computing System, Amoeba, V, and Eden.

1. INTRODUCTION

Everyone agrees that distributed systems are going to be very important in the future. Unfortunately, not everyone agrees on what they mean by the term "distributed system." In this paper we will present a viewpoint widely held within academia about what is and is not a distributed system, discuss numerous interesting design issues concerning them, and finally conclude with a fairly close look at some experimental distributed systems that are the subject of ongoing research at universities.

To begin with, we use the term "distributed system" to mean a distributed *operating system* as opposed to a data base system or some distributed applications system, such as a banking system. An operating system is a program that controls the resources of a computer and provides its users with an interface or virtual machine that is more convenient to use than the bare machine. Examples of well-known centralized (i.e, not distributed) operating systems are: CP/M,[1] MS-DOS,[2] and UNIX.[3]

A *distributed* operating system is one that looks to its users like an ordinary

1. CP/M is a trademark of Digital Research, Inc.
2. MS-DOS is a trademark of Microsoft.
3. UNIX is a trademark of AT&T Bell Laboratories.

The Design of a Capability-Based Distributed Operating System
S. J. MULLENDER and A. S. TANENBAUM
The Computer Journal
Vol. 29, No. 4, pp. 289-300
March 1986

centralized operating system, but runs on multiple, independent CPUs. The key concept here is *transparency,* in other words, the use of multiple processors should be invisible (transparent) to the user. Another way of expressing the same idea is to say that the user views the system as a "virtual uniprocessor," not as a collection of distinct machines. This is easier said than done.

Many multimachine systems that do not fulfill this requirement have been built. For example, the ARPAnet contains a substantial number of computers, but by this definition it is not a distributed system. Neither is a local network consisting of personal computers with minicomputers and explicit commands to log in here or copy a file from there. In both cases we have a computer network but not a distributed operating system. Thus it is the software, not the hardware, that determines whether a system is distributed or not.

As a rule of thumb, if you can tell which computer you are using, you are not using a distributed system. The users of a true distributed system should not know (or care) on which machine (or machines) their programs are running, where their files are stored, and so on. It should be clear by now that very few distributed systems are currently used in a production environment. However, several promising research projects are in progress.

To make the contrast with distributed operating systems stronger, let us briefly look at another kind of system that we will call a *"network operating system."* A typical configuration for a network operating system would be a collection of personal computers along with a common printer server and file server for archival storage, all tied together by a local network. Generally speaking, such a system will have most of the following characteristics that distinguish it from a distributed system:

- Each computer has its own private operating system, rather than running part of a global, system-wide operating system.
- Each user normally works on his own machine; using a different machine invariably requires some kind of "remote login," rather than having the operating system dynamically allocate processes to CPUs.
- Users are typically aware of where each of their files are kept, and must move files between machines with explicit "file transfer" commands, rather than having file placement managed by the operating system.
- The system has little or no fault tolerance; if 1% of the personal computers crash, 1% of the users are out of business, rather than simply having everyone being able to continue normal work, albeit with 1% worse performance.

1.1. GOALS AND PROBLEMS

The driving force behind the current interest in distributed systems is the enormous rate of technological change in microprocessor technology. Microprocessors have become very powerful and cheap, compared to mainframes and minicomputers, so it has become attractive to think about designing large systems composed of many small processors. These distributed systems clearly have a price/performance advantage over more traditional systems. Another

advantage often cited is the relative simplicity of the software - each processor has a dedicated function - although this advantage is more often listed by people who have never tried to write a distributed operating system than those who have.

Incremental growth is another plus; if you need 10% more computing power, you just add 10% more processors. System architecture is crucial to this type of system growth, however, since it is hard to give each user of a personal computer another 10% of a personal computer. Reliability and availability can also be a big advantage; a few parts of the system can be down without disturbing people using the other parts.

On the minus side, unless one is very careful, it is easy for the communication protocol overhead to become a major source of inefficiency. More than one system has been built that required the full computing power of its machines just to run the protocols, leaving nothing over to do the work. The occasional lack of simplicity cited above is a real problem, although in all fairness, this problem comes from inflated goals: with a centralized system no one expects the computer to function almost normally when half the memory is sick. With a distributed system, a high degree of fault tolerance is often, at least, an implicit goal.

A more fundamental problem in distributed systems is the lack of global state information. It is generally a bad idea to even try to collect complete information about any aspect of the system in one table. Lack of up-to-date information makes many things much harder. It is hard to schedule the processors optimally if you are not sure how many are up at the moment.

Despite these obstacles, many people think that in time they can be overcome, so there is great interest in doing research on the subject.

1.2. System Models

Various models have been suggested for building a distributed system. Most of them fall into one of three broad categories, which we will call the "minicomputer" model, the "workstation" model and the "processor pool" model. In the minicomputer model, the system consists of a few (perhaps even a dozen) minicomputers (e.g., VAXes), each with multiple users. Each user is logged onto one specific machine, with remote access to the other machines. This model is a simple outgrowth of the central time-sharing machine.

In the workstation model, each user has a personal workstation, usually equipped with a powerful processor, memory, a bit-mapped display, and sometimes a disk. Nearly all the work is done on the workstations. Such a system begins to look distributed when it supports a single, global file system, so that data can be accessed without regard to its location.

The processor pool model is the next evolutionary step after the workstation model. In a timesharing system, whether with one or more processors, the ratio of CPUs to logged in users is normally much less than 1; with the workstation model it is approximately 1; with the processor pool model it is much greater than 1. As CPUs get cheaper and cheaper, this model will become more and more widespread. The idea here is that whenever a user needs

computing power, one or more CPUs are temporarily allocated to that user; when the job is completed, the CPUs go back into the pool awaiting the next request. As an example, when ten procedures (each on a separate file) must be recompiled, ten processors could be allocated to run in parallel for a few seconds, and then be returned to the pool of available processors. At least one experimental system described below (Amoeba) attempts to combine two of these models, providing each user with a workstation in addition to the processor pool for general use. No doubt other variations will be tried in the future.

2. Network Operating Systems

Before starting our discussion of distributed operating systems, it is worth first taking a brief look at some of the ideas involved in network operating systems, since they can be regarded as primitive forerunners. Although attempts to connect computers together have been around for decades, networking really came into the limelight with the ARPAnet in the early 1970s. The original design did not provide for much in the way of a network operating system. Instead, the emphasis was on using the network as a glorified telephone line to allow remote login and file transfer. Later, several attempts were made to create network operating systems but they never were widely used [Millstein 1977].

In more recent years, several research organizations have connected collections of minicomputers running the UNIX operating system [Ritchie and Thompson 1974] into a network operating system, usually via a local network [Birman and Rowe 1982; Brownbridge et al. 1982; Chesson 1975; Hwang et al. 1982; Wambecq 1983]. Wupit [1983] gives a good survey of these systems, which we will draw upon for the remainder of this section.

As we said earlier, the key issue that distinguishes a network operating system from a distributed one is how aware the users are of the fact that multiple machines are being used. This visibility occurs in three primary areas: the file system, protection, and program execution. Of course it is possible to have systems that are highly transparent in one area and not at all in the other, which leads to a hybrid form.

2.1. File System

When connecting two or more distinct systems together, the first issue that must be faced is how to merge the file systems. Three approaches have been tried. The first approach is not to merge them at all. Going this route means that a program on machine *A* cannot access files on machine *B* by making system calls. Instead, the user must run a special file transfer program that copies the needed remote files to the local machine, where they can then be accessed normally. Sometimes remote printing and mail is also handled this way. One of the best-known examples of networks that primarily support file transfer and mail via special programs, and not system call access to remote files is the UNIX "uucp" program, and its network, USENET.

The next step upward in the direction of a distributed file system is to have **adjoining file systems**. In this approach, programs on one machine can open

files on another machine by providing a path name telling where the file is located. For example, one could say

```
open("/machine1/pathname", READ_ONLY);
open("machine1!pathname", READ_ONLY); or
open("/../machine1/pathname", READ_ONLY)
```

The latter naming scheme is used in the Newcastle Connection [BROWNBRIDGE et al. 1982] and Netix [WAMBECQ 1983] and is derived from the creation of a virtual "superdirectory" above the root directories of all the connected machines. Thus "/.." means start at the local root directory and go upwards one level (to the superdirectory), and then down to the root directory of "machine." In Figure 1, the root directory of three machines, *A*, *B*, and *C* are shown, with a superdirectory above them. To access file *x* from machine *C*, one could say

```
open("/../C/x", READ_ONLY)
```

In the Newcastle system, the naming tree is actually more general, since "machine1" may really be any directory, so one can attach a machine as a leaf anywhere in the hierarchy, not just at the top.

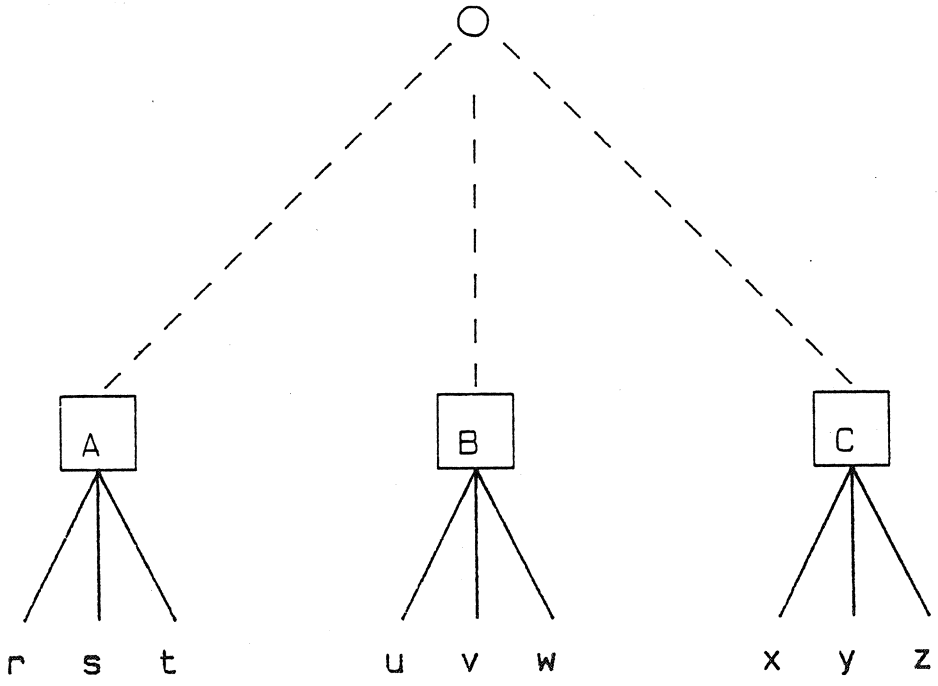


FIGURE 1. A (virtual) superdirectory above the root directory provides access to remote files

The third approach is the way it is done in distributed operating systems, namely to have a single global file system visible from all machines. When this method is used, there is one "bin" directory for binary programs, one password file, and so on. When a program wants to read the password file it does something like

```
open("/etc/passwd", READ_ONLY)
```

without reference to where the file is. It is up to the operating system to locate the file and arrange for transport of data as it is needed. LOCUS is an example of a system using this approach [POPEK et al. 1981; WALKER et al. 1983; WEINSTEIN et al. 1985].

The convenience of having a single global name space is obvious. In addition, this approach means that the operating system is free to move files around between machines to keep all the disks equally full and busy, and that the system can maintain replicated copies of files if it so chooses. When the user or program must specify the machine name, the system cannot decide on its own to move a file to a new machine because that would change the (user visible) name used to access the file. Thus in a network operating system, control over file placement must be done manually by the users, whereas in a distributed operating system it can be done automatically, by the system itself.

2.2. PROTECTION

Closely related to the transparency of the file system is the issue of protection. UNIX, and many other operating systems, assign a unique internal identifier to each user. Each file in the file system has a little table associated with it (called an *i-node* in UNIX), telling who the owner is, where the disk blocks are located, etc. If two previously independent machines are now connected, it may turn out that some internal User IDentifier (UID), e.g., number 12, has been assigned to a different user on each machine. Consequently, when user 12 tries to access a remote file, the remote file system cannot see whether the access is permitted, since two different users have the same UID.

One solution to this problem is require all remote users wanting to access files on machine X to first log onto X using a user name that is local to X. When used this way, the network is just being used as a fancy switch to allow users at any terminal to log onto any computer, just as a telephone company switching center allows any subscriber to call any other subscriber.

This solution is usually inconvenient for people and impractical for programs, so something better is needed. The next step up is to allow any user to access files on any machine without having to log in, but to have the remote user appear to have the UID corresponding to "GUEST" or "DEMO" or some other publicly known login name. Generally such names have little authority, and can only access files that have been designated as readable or writable by all users.

A better approach is to have the operating system provide a mapping between UIDs, so when a user with UID 12 on his home machine accesses a remote machine on which his UID is 15, the remote machine treats all accesses

as though they were done by user 15. This approach implies that sufficient tables are provided to map each user from his home (machine, UID) pair to the appropriate UID for any other machine (and that messages cannot be tampered with)..

In a true distributed system, there should be a unique UID for every user, and that UID should be valid on all machines without any mapping. In this way no protection problems arise on remote accesses to files; as far as protection goes, a remote access can be treated like a local access with the same UID. The protection issue makes the difference between a network operating system and a distributed one clear: in one case there are various machines, each with its own user-to-UID mapping, and in the other there is a single, system-wide mapping that is valid everywhere.

2.3. Execution Location

Program execution is the third area in which machine boundaries are visible in network operating systems. When a user or a running program wants to create a new process, where is the process created? At least four schemes have been used so far. The first of these is that the user simply says "CREATE PROCESS" in one way or another, and specifies nothing about where. Depending on the implementation, this can be the best way or the worst way to do it. In the most distributed case, the system chooses a CPU by looking at the load, location of files to be used, etc. In the least distributed case, the system always runs the process on one specific machine (usually the machine on which the user is logged in).

The second approach to process location is to allow users to run jobs on any machine by first logging in there. In this model, processes on different machines cannot communicate or exchange data, but a simple manual load balancing is possible.

The third approach is special command that the user types at a terminal to cause a program to be executed on a specific machine. A typical command might be

```
remote vax4 who
```

to run the *who* program on machine *vax4*. In this arrangement, the environment of the new process is the remote machine. In other words, if that process tries to read or write files from its current working directory, it will discover that its working directory is on the remote machine, and files that were in the parent process' directory are no longer present. Similarly, files written in the working directory will appear on the remote machine, not the local one.

The fourth approach is to provide the "CREATE PROCESS" system call with a parameter specifying where to run the new process, possibly with a new system call for specifying the default site. As with the previous method, the environment will generally be the remote machine. In many cases, signals and other forms of interprocess communication between processes do not work properly between processes on different machines.

A final point about the difference between network and distributed

operating systems is how they are implemented. A common way to realize a network operating system is to put a layer of software on top of the native operating systems of the individual machines (e.g., MAMRAK et al. 1982). For example, one could write a special library package that intercepted all the system calls and decided whether each one was local or remote [BROWNBRIDGE et al. 1982]. Although most system calls can be handled this way without modifying the kernel, invariably there are a few things, such as interprocess signals, interrupt characters (e.g., BREAK) from the keyboard, etc. that are hard to get right. In a true distributed operating system, one would normally write the kernel from scratch.

2.4. AN EXAMPLE: THE SUN NETWORK FILE SYSTEM

To provide a contrast with the true distributed systems described later in this paper, in this section we will look briefly at a network operating system that runs on the Sun Microsystems' workstations. These workstations are intended for use as personal computers. Each one has a 68000 series CPU, local memory, and a large bitmapped display. Workstations can be configured with or without local disk, as desired. All the workstations run a version of 4.2BSD UNIX specially modified for networking.

This arrangement is a classic example of a network operating system: Each computer runs a traditional operating system, UNIX, and each has its own user(s), but with extra features added to make networking more convenient. During its evolution, the Sun system has gone through three distinct versions, which we will now describe.

In the first version, each of the workstations was completely independent from all the others, except that a program *rcp* was provided to copy files from one workstation to another. By typing a command such as:

```
rcp machine1:/usr/jim/file.c machine2:/usr/ast/f.c
```

it was possible to transfer whole files from one machine to another.

In the second version, Network Disk (ND), a network disk server was provided to support diskless workstations. Disk space on the disk server's machine was divided into disjoint partitions, with each partition acting as the virtual disk for some (diskless) workstation.

Whenever a diskless workstation needed to read a file, the request was processed locally until it got down to the level of the device driver, at which point the block needed was retrieved by sending a message to the remote disk server. In effect, the network was merely being used to simulate a disk controller. With this network disk system, sharing of disk partitions was not possible.

The third version, the Network File System (NFS), allows remote directories to be mounted in the local file tree on any workstation. By mounting, say, a remote directory "doc" on the empty local directory "/usr/doc," all subsequent references to "/usr/doc" are automatically routed to the remote system. Sharing is allowed in NFS, so several users can read files on a remote machine at the same time.

To prevent users from reading other people's private files, a directory can

only be mounted remotely if it is explicitly exported by the workstation it is located on. A directory is exported by entering a line for it in a file "/etc/exports." To improve performance of remote access, both the client machine and server machine do block caching. Remote services can be located using a Yellow Pages server that maps service names onto their network locations.

The NFS is implemented by splitting the operating system up into three layers. The top layer handles directories, and maps each path name onto a generalized i-node called a *vnode* consisting of a (machine, i-node) pair, making each vnode globally unique.

Vnode numbers are presented to the middle layer, the virtual file system (VFS). This layer checks to see if a requested vnode is local or not. If it is local, it calls the local disk driver, or in the case of a ND partition, sends a message to the remote disk server. If it is remote, the VFS calls the bottom layer with a request to process it remotely.

The bottom layer accepts requests for accesses to remote vnodes and sends them over the network to the bottom layer on the serving machine. From there they propagate upwards through the VFS layer to the top layer, where they are re-injected into the VFS layer. The VFS layer sees a request for a local vnode, and processes it normally, without realizing that the top layer is actually working on behalf of a remote kernel. The reply retraces the same path in the other direction.

The protocol between workstations has been carefully designed to be robust in the face of network and server crashes. Each request completely identifies the file (by its vnode), the position in the file, and the byte count. Between requests, the server does not maintain any state information about which files are open or where the current file position is. Thus, if a server crashes and is rebooted, there is no state information that will be lost.

The ND and NFS facilities are quite different, and can both be used on the same workstation without conflict. ND works at a low level and just handles remote block I/O without regard to the structure of the information on the disk. NFS works at a much higher level, and effectively takes requests appearing at the top of the operating system on the client machine and gets them over to the top of the operating system on the server machine, where they are processed the same way as local requests.

3. Design Issues

Now we turn from traditional computer systems with some networking facilities added on to systems designed with the intention of being distributed. In this section we will look at five issues that distributed systems' designers are faced with:

- communications primitives,
- naming and protection,
- resource management,
- fault tolerance,
- services to provide.

While no list could possibly be exhaustive at this early stage of development, these topics should provide a reasonable impression of the areas in which current research is proceeding.

3.1. Communication Primitives

The computers forming a distributed system normally do not share primary memory, so communication via shared memory techniques such as semaphores and monitors are generally not applicable. Instead, message passing in one form or another is used. One widely discussed framework for message-passing systems is the ISO OSI reference model, which has seven layers, each performing a well-defined function [Zimmerman 1980]. The seven layers are: physical layer, data-link layer, network layer, transport layer, session layer, presentation layer, and application layer. Using this model it is possible to connect computers with widely different operating systems, character codes, and ways of viewing the world.

Unfortunately, the overhead created by all these layers is substantial. In a distributed system consisting primarily of huge mainframes from different manufacturers, connected by slow leased lines (say, 56 kbps), the overhead might be tolerable. Plenty of computing capacity would be available for running complex protocols, and the narrow bandwidth means that close coupling between the systems would be impossible anyway. On the other hand, in a distributed system consisting of identical microcomputers connected by a 10 Mbps or faster local network, the price of the ISO model is generally too high. Nearly all the experimental distributed systems discussed in the literature so far have opted for a different, much simpler model, so we will not mention the ISO model further in this paper.

3.1.1 Message Passing

The model that is favored by researchers in this area is the **client-server model**, in which a client process wanting some service (e.g., reading some data from a file) sends a message to the server and then waits for a reply message, as shown in Figure 2. In the most naked form, the system just provides two primitives: SEND and RECEIVE. The SEND primitive specifies the destination and provides a message; the RECEIVE primitive tells from whom a message is desired (including "anyone") and provides a buffer where the incoming message is to be stored. No initial setup is required, and no connection is established, hence no teardown is required.

Precisely what semantics these primitives ought to have has been a subject of much controversy among researchers. Two of the fundamental decisions that must be made are unreliable vs. reliable and nonblocking vs. blocking

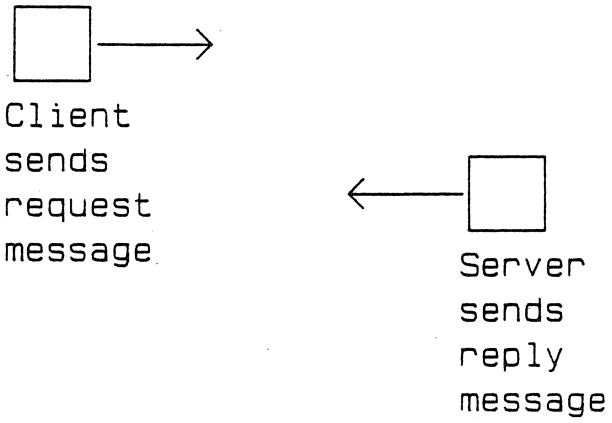


FIGURE 2. Client-server model of communication.

primitives. At one extreme, SEND can put a message out onto the network and wish it good luck. No guarantee of delivery is provided, and no automatic retransmission is attempted by the system if the message is lost. At the other extreme, SEND can handle lost messages, retransmissions, and acknowledgements internally, so that when SEND terminates, the program is sure that the message has been received and acknowledged.

Blocking vs. Nonblocking Primitives. The other choice is between nonblocking and blocking primitives. With nonblocking primitives, SEND returns control to the user program as soon as the message has been queued for subsequent transmission (or a copy made). If no is copy is made, any changes the program makes to the data before or (heaven forbid) *while* it is being sent, are made at the program's peril. When the message has been transmitted (or copied to a safe place for subsequent transmission), the program is interrupted to inform it that the buffer may be reused. The corresponding RECEIVE primitive signals a willingness to receive a message, and provides a buffer for it to be put into. When a message has arrived, the program is informed by interrupt or it can poll for status continuously, or go to sleep until the interrupt arrrives. The advantage of these nonblocking primitives is that they provide the maximum flexibility: programs can compute and perform message I/O in parallel any way they want to.

Nonblocking primitives also have a disadvantage: they make programming tricky and difficult. Irreproducible, timing-dependent programs are painful to write and awful to debug. Consequently, many people advocate sacrificing some flexibility and efficiency by using blocking primitives. A blocking SEND does not return control to the user until the message has been sent (unreliable blocking primitive) or until the message has been sent and an acknowledgement received (reliable blocking primitive). Either way, the program may immediately modify the buffer without danger. A blocking RECEIVE does

not return control until a message has been placed in the buffer. Reliable and unreliable RECEIVEs differ in that the former automatically acknowledges receipt of message, whereas the latter does not. It is not reasonable to combine a reliable SEND with an unreliable RECEIVE or vice versa, so the system designers must make a choice and provide one set or the other. Blocking and nonblocking primitives do not conflict, so there is no harm done if the sender uses one and the receiver the other.

Buffered vs. Unbuffered Primitives. Another design decision that must be made is whether or not to buffer messages. The simplest strategy is not to buffer. When a sender has a message for a receiver that has not (yet) executed a RECEIVE primitive, the sender is blocked until a RECEIVE has been done, at which time the message is copied from sender to receiver. This strategy is sometimes referred to as a **rendezvous**.

A slight variation on this theme is to copy the message to an internal buffer on the sender's machine, thus providing for a nonblocking version of the same scheme. As long as the sender does not do any more SENDs before the RECEIVE occurs, no problem occurs.

A more general solution is to have a buffering mechanism, usually in the operating system kernel, which allows senders to have multiple SENDs outstanding even without any interest on the part of the receiver. Although buffered message passing can be implemented in many ways, a typical approach is to provide users with a system call CREATEBUF, which creates a kernel buffer, sometimes called a **mailbox**, of a user-specified size. To communicate, a sender can now send messages to the receiver's mailbox, where they will be buffered until requested by the receiver. Buffering is not only more complex (creating, destroying, and generally managing the mailboxes), but also raises issues of protection, the need for special high-priority interrupt messages, what to do with mailboxes owned by processes that have been killed or died of natural causes, and more.

A more structured form of communication is achieved by distinguishing requests from replies. With this approach, one typically has three primitives: SEND_GET, GET_REQUEST, and SEND_REPLY. SEND_GET is used by clients to send requests and get replies. It combines a SEND to a server with a RECEIVE to get the server's reply. GET_REQUEST is done by servers to acquire messages containing work for them to do. When a server has carried the work out, it sends a reply with SEND_REPLY. By thus restricting the message traffic, and by using reliable, blocking primitives, one can create some order in the chaos.

3.1.2. Remote Procedure Call

The next step forward in message-passing systems is the realization that the model of "client sends request and blocks until server sends reply" looks very similar to a traditional procedure call from the client to the server. This model has become known in the literature as "**remote**procedure and has been widely discussed [BIRRELL and NELSON 1984; NELSON 1981; SPECTOR 1982]. The idea

is to make the semantics of intermachine communication as similar as possible to normal procedure calls because the latter is familiar, well understood, and has proved its worth over the years as a tool for dealing with abstraction. It can be viewed as a refinement of the reliable, blocking SEND_GET, GET_REQUEST, SENDREP primitives, with a more user-friendly syntax.

The remote procedure call can be organized as follows. The client (calling program) makes a normal procedure call, say, *p(x, y)* on its machine, with the intention of invoking the remote procedure *p* on some other machine. A dummy or **stub** procedure *p* must be included in the caller's address space, or at least be dynamically linked to it upon call. This procedure, which may be automatically generated by the compiler, collects the parameters and packs them into a message in a standard format. It then sends the message to the remote machine (using SEND_GET) and blocks, waiting for an answer (see Figure 3).

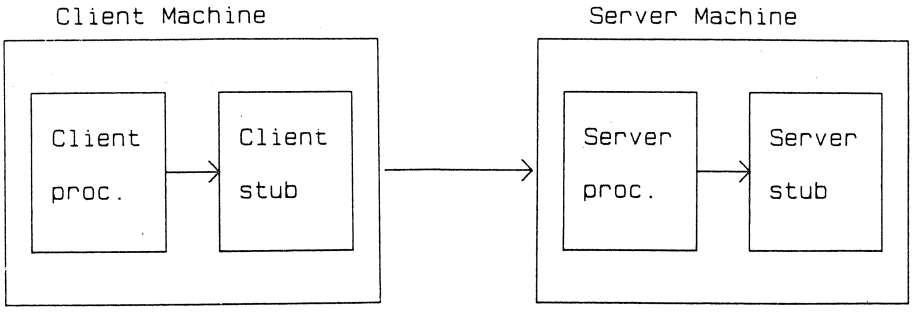


FIGURE 3. Remote procedure call.

At the remote machine, another stub procedure should be waiting for a message using GET_REQUEST. When a message comes in, the parameters are unpacked by an input handling procedure, which then makes the local call *p(x, y)*. The remote procedure *p* is thus called locally, so its normal assumptions about where to find parameters, the state of the stack, etc., are identical to the case of a purely local call. The only procedures that know that the call is remote are the stubs, which build and send the message on the client side and disassemble and make the call on the server side. The result of the procedure call follows an analogous path in the reverse direction.

RPC Design Issues. Although at first glance the remote procedure call model seems clean and simple, under the surface there are several problems. One problem concerns parameter (and result) passing. In most programming languages, parameters can be passed by value or by reference. Passing value parameters over the network is easy; the stub just copies them into the message and off it goes. Passing reference parameters (pointers) over the network is not so easy. One needs a unique, system-wide pointer for each object so that it can be remotely accessed. For large objects, such as files, some kind of capability mechanism [DENNIS and VAN HORN 1966; LEVY 1984; PASHTAN 1982] could be set up, using capabilities as pointers. For small objects, such as integers and booleans, the amount of overhead and mechanism needed to create a capability and send it in a protected way is so large that this solution is highly undesirable.

Still another problem that must be dealt with is how to represent parameters and results in messages. This representation is greatly complicated when different types of machines are involved in a communication. A floating-point number produced on one machine is unlikely to have the same value on a different machine, and even a negative integer will create problems between 1's and 2's complement machines.

Converting to and from a standard format on every message sent and received is an obvious possibility, but it is expensive and wasteful, especially when the sender and receiver do, in fact, use the same internal format. If the sender uses its internal format (along with an indication of which format it is) and let the receiver do the conversion, every machine must be prepared to convert from every other format. When a new machine type is introduced, much existing software must be upgraded. Any way you do it, with RPC or with plain messages, it is an unpleasant business.

Some of the unpleasantness can be hidden from the user if the remote procedure call mechanism is embedded in a programming language with strong typing, so at least the receiver knows how many parameters to expect and what types they have. In this respect, a weakly-typed language such as C, in which procedures with a variable number of parameters are common, is more complicated to deal with.

Still another problem with RPC is the issue of client-server binding. Consider, for example, a system with multiple file servers. If a client creates a file on one of the file servers, it is usually desirable that subsequent writes to that file to go to the file server where the file was created. With mailboxes, arranging for this is straightforward. The client simply addresses the WRITE messages to the same mailbox that the CREATE message was sent to. Since each file server has its own mailbox, there is no ambiguity.

When RPC is used, the situation is more complicated, since all the client does is put a procedure call such as

write(FileDescriptor, BufferAddress, ByteCount);

in his program. RPC intentionally hides all the details of locating servers from

the client, but sometimes, as in this example, the details are important.

In some applications, broadcasting and multicasting (sending to a set of destinations, rather than just one is useful. For example, when trying to locate a certain person, process, or service, sometimes the only approach is to broadcast an inquiry message and wait for the replies to come back. RPC does not lend itself well to sending messages to sets of processes and getting answers back from some or all of them. The semantics are completely different.

Despite all these problems, RPC remains an interesting form of communication, and much current research is being addressed to improving it and solving the various discussed above.

3.1.3. Error Handling

In error handling, the communication primitives of distributed systems differ radically from those of centralized systems. In a centralized system, a system crash means that the client, server, and communication channel are all completely destroyed, and no attempt is made to revive them. In a distributed system, matters are more complex. If a client has initiated a remote procedure call with a server that has crashed, the client may just be left hanging forever unless a timeout is built in. However, such a timeout introduces race conditions in the form of clients that time out too quickly, thinking that the server is down, when in fact, it is merely very slow.

Client crashes can also cause trouble for servers. Consider for example, the case of processes *A* and *B* communicating via the UNIX pipe model *A*|*B* with *A* the server and *B* the client. *B* asks *A* for data and gets a reply, but unless that reply is acknowledged somehow, *A* does not know when it can safely discard data that it may not be able to reproduce. If *B* crashes, how long should *A* hold onto the data? (Hint: if the answer is less than infinity, problems will be introduced whenever *B* is slow in sending an acknowledgement.)

Closely related to this is the problem of what happens if a client cannot tell whether or not a server has crashed. Simply waiting until the server is rebooted and trying again sometimes works and sometimes does not. A case where it works: client asks to read block 7 of some file. A case where it does not work: client says transfer a million dollars from one bank account to another. In the former case, it does not matter whether or not the server carried out the request before crashing; carrying it out a second time does no harm. In the latter case, one would definitely prefer the call to be carried out exactly once, no more and no less. Calls that may be repeated without harm (like the first example) are said to be **idempotent**. Unfortunately, it is not always possible to arrange for all calls to have this property. Any call that causes action to occur in the outside world, such as transferring money, printing lines, or opening a valve in an automated chocolate factory just long enough to fill exactly one vat, is likely to cause trouble if performed twice.

SPECTOR [1982] and NELSON [1981] have looked at the problem of trying to make sure remote procedure calls are executed exactly once, and have developed taxonomies for classifying the semantics of different systems. These vary from systems that offer no guarantee at all (zero or more executions), to

those that guarantee at most one execution (zero or one), to those that guarantee at least one execution (one or more).

Getting it right (exactly one) is probably impossible, because even if the remote execution can be reduced to one instruction (e.g., setting a bit in a device register that opens the chocolate valve), one can never be sure after a crash if the system went down a microsecond before, or a microsecond after, the one critical instruction. Sometimes one can make a guess based on observing external events (e.g., looking to see if the factory floor is covered with a sticky, brown material), but in general there is no way of knowing. Note that the problem of creating stable storage [LAMPSON 1981] is fundamentally different, since remote procedure calls to the stable storage server in that model never causes events external to the computer.

3.1.4. Implementation Issues

Constructing a system in principle is always easier than constructing it in practice. Building a 16-node distributed system that has a total computing power about equal to a single-node system is surprisingly easy. This observation leads to tension between the goals of making it work fast in the normal case, and making the semantics reasonable when something goes wrong. Some experimental systems have put the emphasis on one goal and some on the other, but more research is needed before we have systems that are both fast and graceful in the face of crashes.

Some things have been learned from past work, however. Foremost among these is that making message passing efficient is very important. To this end, systems should be designed to minimize copying of data [CHERITON 1984]. For example, a remote procedure call system that first copies each message from the user to the stub, and then from the stub to the kernel, and finally from the kernel to the network interface board requires 3 copies on the sending side, and probably 3 more on the receiving side, for a total of 6. If the call is to a remote file server to write a 1K block of data to disk, at a copy time of 1 microsec per byte, 6 msec are needed just for copying, which puts an upper limit of 167 calls/sec, or a throughput of 167 Kbytes/sec. When other sources of overhead are considered (e.g., the reply message, the time waiting for access to the network, transmission time, etc.) achieving even 80 Kbytes/sec will be difficult, if not impossible, no matter how high the network bandwidth or disk speed. Thus, it is desirable to avoid copying, but this is not always simple to achieve since without copies, (part of) a needed message may be swapped or paged out when it is needed.

Another point worth making is that there is always a substantial fixed overhead with preparing, sending, and receiving a message, even a short message, such as a request to read from a remote file server. The kernel must be invoked, the state of the current process must be saved, the destination must be located, various tables must be updated, permission to access the network must be obtained (e.g., wait for the network to become free or wait for the token), and quite a bit of bookkeeping must be done.

This fixed overhead argues for making messages as long as possible, to

reduce the number of messages. Unfortunately. many current local networks limit physical packets to 1K or 2K; 4K or 8K would be much better. Of course, if the packets become too long, a highly interactive user may occasionally be queued behind 10 maximum length packets, degrading response time, so the optimum size depends on the work load.

Virtual Circuits vs. Datagrams. There is much controversy over whether remote procedure call ought to be built on top of a flow-controlled, error-controlled, virtual circuit mechanism, or directly on top of the unreliable, connectionless (datagram) service. SALTZER et al. [1984] have pointed out that since high reliability can only be achieved by end-to-end acknowledgements at the highest level of protocol, the lower levels need not be 100% reliable. The overhead incurred in providing a clean virtual circuit upon which to build remote procedure calls (or any other message passing system), is therefore wasted. This line of thinking argues for building the message system directly on the raw datagram interface.

The other side of the coin is that it would be nice for a distributed system to be able to encompass heterogeneous computers in different countries with different PTT networks and possibly different national alphabets, and that this environment requires complex multilayered protocol structures. It is our observation that both arguments are valid, but depending on whether one is trying to forge a collection of small computers into a virtual uniprocessor or merely access remote data transparently, one or the other will dominate.

Even if one chooses for building remote procedure call on top of the raw datagram service provided by a local network, there are still a number of protocols open to the implementer. The simplest one is to have every request and reply separately acknowledged. The message sequence for a remote procedure call is then: REQUEST, ACK, REPLY, ACK, as shown in The ACKs are managed by the kernel without user knowledge.

The number of messages can be reduced from four to three by allowing the REPLY to serve as the ACK for the REQUEST, as shown in Figure 4. However, a problem arises when the REPLY can be delayed for a long time. For example, when a login process makes a remote procedure call to a terminal server requesting characters, it may be hours or days before someone steps up to a terminal and begins typing. In this event, an additional message has to be introduced to allow the sending kernel to inquire if the message arrived or not.

A further step in the same direction is to eliminate the other ACK as well, and let the arrival of the next REQUEST imply an acknowledgement of the previous REPLY (see Figure 4(c)). Again, some mechanism is needed to deal with the case that no new REQUEST is forthcoming quickly.

One of the great difficulties in implementing efficient communication is that it is more of a black art than a science. Even straightforward implementations can have unexpected consequences, as the following example from SVENTEK et al. [1983] shows. Consider a ring containing a circulating token. To transmit, a machine captures and removes the token, puts a message on the network, and then replaces the token, thus allowing the next machine "downstream" the

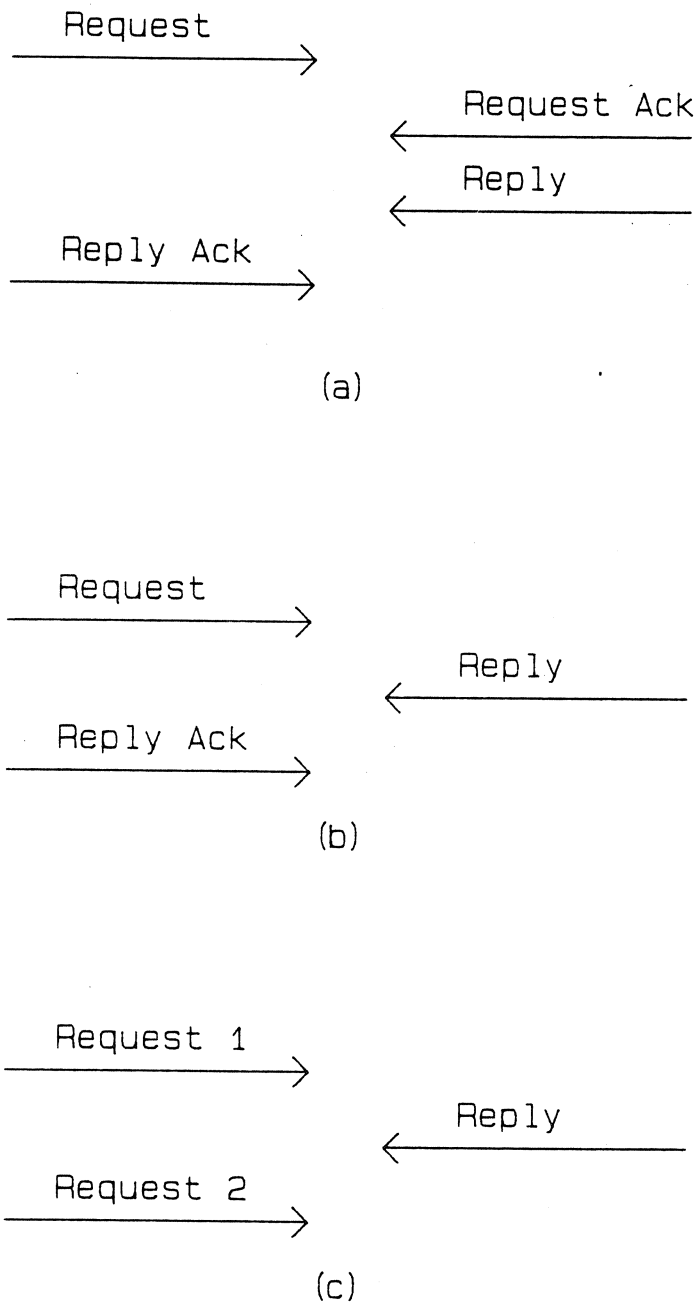


FIGURE 4. Remote procedure call (a) with individual acknowledgements per message, (b) with the reply as the request acknowledgement, (c) with no explicit acknowledgements.

opportunity to capture it. In theory, such a network is "fair" in that each user has equal access to the network and no one user can monopolize it to the detriment of others. In practice, suppose two users each want to read a long file from a file server. User *A* sends a request message to the server, and then replaces the token on the network for *B* to acquire.

After *A*'s message arrives at the server, it takes a short time for the server to handle the incoming message interrupt and re-enable the receiving hardware. Until the receiver is re-enabled, the server is deaf. Within a microsecond or two of the time *A* puts the token back on the network, *B* sees and grabs it, and begins transmitting a request to the (unbeknownst to *B*) deaf file server. Even if the server re-enables halfway through *B*'s message, the message will be rejected due to missing header, bad frame format, and checksum error. According to the ring protocol, after sending one message, *B* must now replace the token, which *A* captures for a successful transmission. Once again *B* transmits during the server's deaf period, and so on. Conclusion: *B* gets no service at all until *A* is finished. If *A* happens to be scanning through the Manhattan telephone book, *B* may be in for a long wait. This specific problem can be solved by inserting random delays in places to break the synchrony, but our point is that totally unexpected problems like this make it necessary to build and observe real systems to gain insight into the problems. Abstract formulations and simulations are not enough.

3.2. Naming and Protection

All operating systems support objects such as files, directories, segments, mailboxes, processes, services, servers, nodes, and I/O devices. When a process wants to access one of these objects, it must present some kind of name to the operating system to specify which object it wants to access. In some instances these names are ASCII strings designed for human use, in others they are binary numbers used only internally. In all cases they have to be managed and protected from misuse.

3.2.1. Naming as Mapping

Naming can best be seen as a problem of mapping between two domains. For example, the directory system in UNIX provides a mapping between ASCII path names and i-node numbers. When an OPEN system call is made, the kernel converts the name of the file to be opened into its i-node number. Internal to the kernel, files are nearly always referred to by i-node number, not ASCII string. Just about all operating systems have something similar. In a distributed system a separate name server is sometimes used to map user-chosen names (ASCII strings) onto objects in an analogous way.

Another example of naming is the mapping of virtual addresses onto physical addresses in a virtual memory system. The paging hardware takes a virtual address as input, and yields a physical address as output for use by the real memory.

In some cases naming implies only a single level of mapping, but in other cases it can imply multiple levels. For example, to use some service, a process

might first have to map the service name onto the name of a server process that is prepared to offer the service. As a second step, the server would then be mapped onto the number of the CPU on which it that process is running. The mapping need not always be unique, for example, if there are multiple processes prepared to offer the same service.

3.2.2. Name Servers

In centralized systems, the problem of naming can be effectively handled in a straightforward way. The system maintains a table or data base providing the necessary name-to-object mappings. The most straightforward generalization of this approach to distributed systems is the single name server model. In this model, a server accepts names in one domain and maps them onto names in another domain. For example, to locate services in some distributed systems, one sends the service name in ASCII to the name server, and it replies with the node number where that service can be found, or with the process name of the server process, or perhaps with the name of a mailbox to which requests for service can be sent. The name server's data base is built up by registering services, processes, etc., that want to be publicly known. File directories can be regarded as a special case of name service.

Although this model is often acceptable in a small distributed system located at a single site, in a large system it is undesirable to have a single centralized component (the name server) whose demise can bring the whole system to a grinding halt. In addition, if it becomes overloaded, performance will degrade. Furthermore, in a geographically distributed system that may have nodes in different cities or even countries, having a single name server will be inefficient due to the long delays in accessing it.

The next approach is to partition the system into domains, each with its own name server. If the system is composed of multiple local networks connected by gateways and bridges, it seems natural to have one name server per local network. One way to organize such a system is to have a global naming tree, with files and other objects having names of the form: /country/city/network/pathname. When such a name is presented to any name server, it can immediately route the request to some name server in the designated country, which then sends it to a name server in the designated city, and so on until it reaches the name server in the network where the object is located, where the mapping can be done. Telephone numbers use such a hierarchy, composed of country code, area code, exchange code (first 3 digits of telephone number in North America), and subscriber line number.

Having multiple name servers does not necessarily require having a single, global naming hierarchy. Another way to organize the name servers is to have each one effectively maintain a table of, for example, (ASCII string, pointer) pairs, where the pointer is really a kind of capability for any object or domain in the system. When a name, say *a/b/c*, is looked up by the local name server, it may well yield a pointer to another domain (name server) , to which the rest of the name, *b/c*, is sent for further processing (see Figure 5). This facility can be used to provide links (in the UNIX sense) to files or objects

whose precise whereabouts is managed by a remote name server. Thus if a file *foobar* is located in another local network, *n*, with name server *n.s*, one can make an entry in the local name server's table for the pair (*x*, *n.s*) and then access *x/foobar* as though it were a local object. Any appropriately authorized user or process knowing the name *x/foobar* could make its own synonym *s* and then perform accesses using *s/x/foobar*. Each name server parsing a name that involves multiple name servers just strips off the first component and passes the rest of the name to the name server found by looking up the first component locally.

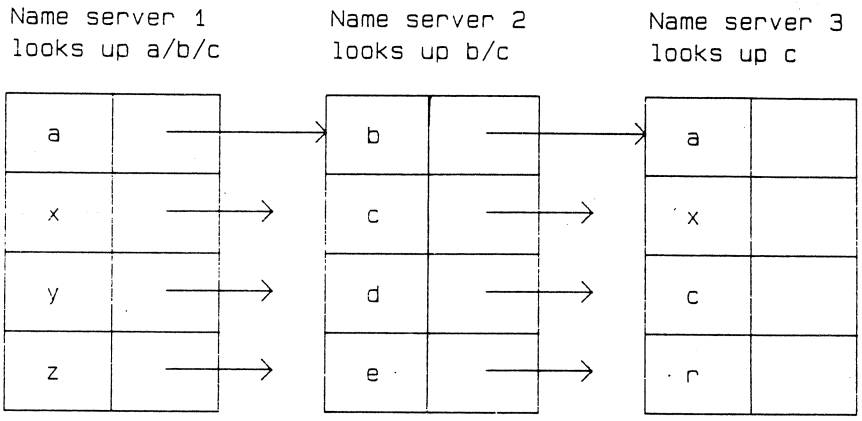


FIGURE 5. Distributing the lookup of a/b/c over three name servers.

A more extreme way of distributing the name server is to have each machine manage its own names. To look up a name, one broadcasts it on the network. At each machine, the incoming request is passed to the local name server, which replies only if it finds a match. Although broadcasting is easiest over a local network such as a ring net or CSMA net (e.g., Ethernet), it is also possible over store-and-forward packet switching networks such as the ARPAnet [DALAL 1977].

Although the normal use of a name server is to map an ASCII string onto a binary number used internally to the system, such as a process identifier or machine number, once in a while the inverse mapping is also useful. For example, if a machine crashes, upon rebooting it could present its (hardwired) node number to the name server to ask what it was doing before the crash, that is, ask for the ASCII string corresponding to the service it is supposed to be offering so it can figure out what program to reboot.

3.3. Resource Management

Resource management in a distributed system differs from that in a centralized system in a fundamental way. Centralized systems always have tables that give complete and up-to-date status information about all the resources being managed; distributed systems do not. For example, the process manager in a traditional centralized operating system normally uses a "process table" with one entry per potential process. When a new process has to be started, it is simple enough to scan the whole table to see if a slot is free. A distributed operating system, on the other hand, has a much harder job of finding out if a processor is free, especially if the system designers have rejected the idea of having any central tables at all, for reasons of reliability. Furthermore, even if there is a central table, recent events on outlying processors may have made some table entries obsolete without the table manager knowing it.

The problem of managing resources without having accurate global state information is very difficult. Relatively little work has been done in this area. In the following sections we will look at some work that has been done, including distributed process management and scheduling.

3.3.1. Processor Allocation

One of the key resources to be managed in a distributed system is the set of available processors. One approach that has been proposed for keeping tabs on a collection of processors is to organize them in a logical hierarchy independent of the physical structure of the network, as in MICROS [Wittie and van Tilborg 1980]. This approach organizes the machines like people in corporate, military, academic, and other real-world hierarchies. Some of the machines are workers and others are managers.

For each group of k workers, one manager machine (the "department head") is assigned the task of keeping track of who is busy and who is idle. If the system is large, there will be an unwieldy number of department heads, so some machines will function as "deans," riding herd on k department heads. If there are many deans, they too can be organized hierarchically, with a "big cheese" keeping tabs on k deans. This hierarchy can be extended ad infinitum, with the number of levels needed growing logarithmically with the number of workers. Since each processor need only maintain communication with one superior and k subordinates, the information stream is manageable.

An obvious question is "What happens when a department head, or worse yet, a big cheese, stops functioning (crashes)?" One answer is to promote one of the direct subordinates of the faulty manager to fill in for the boss. The choice of which one can either be made by the subordinates themselves, by the deceased's peers, or in a more autocratic system, by the sick manager's boss.

To avoid having a single (vulnerable) manager at the top of the tree, one can truncate the tree at the top and have a committee as the ultimate authority. When a member of the ruling committee malfunctions, the remaining members promote someone one level down as replacement.

While this scheme is not completely distributed, it is feasible, and in practice works well. In particular, the system is self-repairing, and can survive

occasional crashes of both workers and managers without any long-term effects.

In MICROS, the processors are monoprogrammed, so if a job requiring S processes suddenly appears, the system must allocate S processors for it. Jobs can be created at any level of the hierarchy. The strategy used is for each manager to keep track of approximately how many workers below it are available (possibly several levels below it). If it thinks that a sufficient number are available, it reserves some number R of them, where $R >= S$, because the estimate of available workers may not be exact and some machines may be down.

If the manager receiving the request thinks that it has too few processors available, it passes the request upwards in the tree to its boss. If the boss cannot handle it either, the request continues propagating upward until it reaches a level that has enough available workers at its disposal. At that point, the manager splits the request into parts, and parcels them out among the managers below it, which then do the same thing until the wave of scheduling requests hits bottom. At the bottom level, the processors are marked as "busy" and the actual number of processors allocated is reported back up the tree.

To make this strategy work well, R must be large enough that the probability is high that enough workers will be found to handle the whole job. Otherwise the request will have to move up one level in the tree and start all over, wasting considerable time and computing power. On the other hand, if R is too large, too many processors will be allocated, wasting computing capacity until word gets back to the top and they can be released.

The whole situation is greatly complicated by the fact that requests for processors can be generated randomly anywhere in the system, so at any instant, multiple requests are likely to be in various stages of the allocation algorithm, potentially giving rise to out-of-date estimates of available workers, race conditions, deadlocks, and more. In [VAN TILBORG and WITTIE 1981] a mathematical analysis of the problem is given and various other aspects not described here are covered in detail.

3.3.2. Scheduling

The hierarchical model provides a general model for resource control, but does not provide any specific guidance on how to do scheduling. If each process uses an entire processor (i.e., no multiprogramming), and each process is independent of all the others, any process can be assigned to any processor at random. However, if it is common that several processes are working together and must communicate frequently with each other, as in UNIX pipelines or in cascaded (nested) remote procedure calls, then it is desirable to make sure the whole group runs at once. In this section we will address that issue.

Let us assume that each processor can handle up to N processes. If there are plenty of machines and N is reasonably large, the problem is not finding a free machine (i.e., a free slot in some process table), but something more subtle. The basic difficulty can be illustrated by an example in which processes A

and *B* run on one machine and processes *C* and *D* run on another. Each machine is time-shared in, say, 100 msec time slices, with *A* and *C* running in the even slices, and *B* and *D* running in the odd ones, as shown in Figure 6(a). Suppose that *A* sends many messages or makes many remote procedure calls to *D*. During time slice 0, *A* starts up and immediately calls *D*, which unfortunately is not running because it is now *C*'s turn. After 100 msec, process switching takes place, and *D* gets *A*'s message, carries out the work, and quickly replies. Because *B* is now running, it will be another 100 msec before *A* gets the reply and can proceed. The net result is one message exchange every 200 msec. What is needed is a way to ensure that processes that communicate frequently run simultaneously.

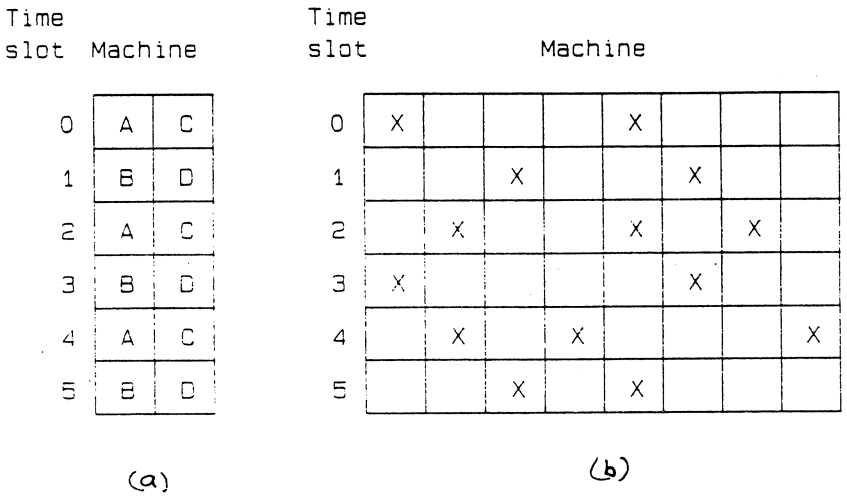


FIGURE 6. (a) Two jobs running out of phase with each other. (b) Scheduling matrix for 8 machines, each with six time slots. The *X*'s indicated allocated slots.

Although it is difficult to dynamically determine the interprocess communication patterns, in many cases, a group of related processes will be started off together. For example, it is usually a good bet that the filters in a UNIX pipeline will communicate with each other more than they will with other, previously started processes. Let us assume that processes are created in groups, and that intragroup communication is much more prevalent than intergroup communication. Let us further assume that a sufficiently large number of machines is available to handle the largest group, and that each machine is multiprogrammed with *N* process slots (*N*-way multiprogramming).

OUSTERHOUT [1982] has proposed several algorithms based on the concept of *co-scheduling*, which takes interprocess communication patterns into account while scheduling to ensure that all members of a group run at the same time. The first algorithm uses a conceptual matrix in which each column is the

process table for one machine, as shown in Figure 6(b). Thus, column 4 consists of all the processes that run on machine 4. Row 3 is the collection of all processes that are in slot 3 of some machine, starting with the process in slot 3 of machine 0, then the process in slot 3 of machine 1, and so on. The gist of his idea is to have each processor use a round robin scheduling algorithm with all processors first running the process in slot 0 for a fixed period, then all processors running the process in slot 1 for a fixed period, etc. A broadcast message could be used to tell each processor when to do process switching, to keep the time slices synchronized.

By putting all the members of a process group in the same slot number, but on different machines, one has the advantage of N-fold parallelism, with a guarantee that all the processes will be run at the same time, to maximize communication throughput. Thus in Figure 6(b), four processes that must communicate should be put into slot 3, on machines 1, 2, 3, and 4 for optimum performance. This scheduling technique can be combined with the hierarchical model of process management used in MICROS by having each department head maintain the matrix for its workers, assigning processes to slots in the matrix and broadcasting time signals.

Ousterhout also described several variations to this basic method to improve performance. One of these breaks the matrix into rows, and concatenates the rows to form one long row. With k machines, any k consecutive slots belong to different machines. To allocate a new process group to slots, one lays a window k slots wide over the long row such that the leftmost slot is empty but the slot just outside the left edge of the window is full. If sufficient empty slots are present in the window, the processes are assigned to the empty slots, otherwise the window is slid to the right and the algorithm repeated. Scheduling is done by starting the window at the left edge and moving rightward by about one window's worth per time slice, taking care not to split groups over windows. Ousterhout's paper discusses these and other methods in more detail and gives some performance results.

3.3.3. Load Balancing

The goal of Ousterhout's work is to place processes that work together on different processors, so that they can all run in parallel. Other researchers have tried to do precisely the opposite, namely, to find subsets of all the processes in the system that are working together, so closely related groups of processes can be placed on the same machine to reduce interprocess communication costs [CHU et al, 1980; CHOW and ABRAHAM 1982; GYLYS and EDWARDS 1976; STONE 1977; STONE 1978; STONE and BOKHARI 1978; LO 1984]. Yet other researchers have been concerned primarily with load balancing, to prevent a situation in which some processors are overloaded while others are empty [BARAK and SHILOH 1985; EFE 1982; KRUEGER and FINKEL 1983; STANKOVIC and SIDHU 1984]. Of course, the goals of maximizing throughput, minimizing response time, and keeping the load uniform, are to some extent in conflict, so many of the researchers try to evaluate different compromises and tradeoffs.

Each of these different approaches to scheduling makes different assumptions about what is known and what is most important. The people trying to cluster processes to minimize communication costs, for example, assume that any process can run on any machine, that the computing needs of each process are known in advance, and that the interprocess communication traffic between each pair of processes is also known in advance. The people doing load balancing typically make the realistic assumption that nothing about the future behavior of a process is known. The minimizers are generally theorists, whereas the load balancers tend to be people making real systems who care less about optimality than devising algorithms that can actually be used. Let us now briefly look at each of these approaches.

Graph Theoretic Models. If the system consists of a fixed number of processes, each with known CPU and memory requirements, and a known matrix giving the average amount of traffic between each pair of processes, scheduling can be attacked as a graph-theoretic problem. The system can be represented as a graph, with each process a node, and each pair of communicating processes connected by an arc labeled with the data rate between them.

The problem of allocating all the processes to k processors then reduces to the problem of partitioning the graph into k disjoint subgraphs, such that each subgraph meets certain constraints (e.g., total CPU and memory requirements below some limit). Arcs that are entirely within one subgraph represent internal communication within a single processor (= fast), whereas arcs that cut across subgraph boundaries represent communication between two processors (= slow). The idea is to find a partitioning of the graph that meets the constraints and minimizes the network traffic, or some variation of this idea. Figure 7(a) depicts a graph of interacting processors with one possible partitioning of the processes between two machines. Figure 7(b) shows a better partitioning, with less intermachine traffic, assuming that all the arcs are equally weighted. Many papers have been written on this subject, for example, [CHOW and ABRAHAM 1982; STONE 1977; STONE 1978; STONE and BOKHARI 1978; LO 1984]. The results are somewhat academic, since in real systems virtually none of the assumptions (fixed number of processes with static requirements, known traffic matrix, error-free processors and communication) are ever met.

Heuristic load balancing. When the goal of the scheduling algorithm is dynamic, heuristic, load balancing, rather than finding related clusters, a different approach is taken. Here the idea is for each processor to continually estimate its own load, for processors to exchange load information, and for process creation and migration to utilize this information.

Various methods of load estimation are possible. One way is just to measure the number of runnable processes on each CPU periodically, and take the average of the last n measurements as the load. Another way [BRYANT and FINKEL 1981] is to estimate the residual running times of all the processes and define the load on a processor as the number of CPU seconds all its processes will need to finish. The residual time can be estimated mostly simply by

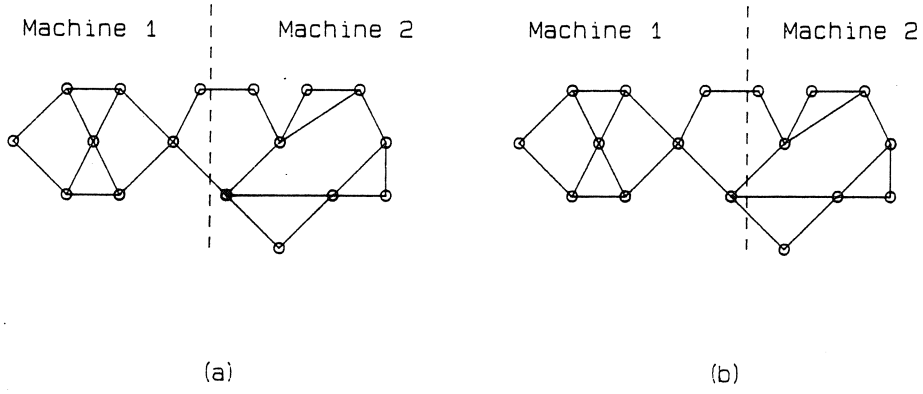


FIGURE 7. Two ways of statically allocating processes (nodes in the graph) to machines. Arcs show which pairs of processes communicate.

assuming it is equal to the CPU time already consumed. Bryant and Finkel also discuss other estimation techniques in which both the number of processes and length of remaining time are important. When round robin scheduling is used, it is better to be competing against one process that needs 100 sec than against 100 processes that each need 1 sec.

Once each processor has computed its load, a way is needed for each processor to find out how everyone else is doing. One way is for each processor to just broadcast its load periodically. After receiving a broadcast from a lightly loaded machine, a processor should shed some of its load by giving it to the lightly loaded processor. This algorithm has several problems. First, it requires a broadcast facility, which may not be available. Second, it consumes considerable bandwidth for all the "Here is my load" messages. Third, there is a great danger that many processors will try to shed load to the same (previously) lightly loaded processor at once.

A different strategy [SMITH 1979; BARAK and SHILOH 1985] is for each processor to periodically pick another processor (possibly a neighbor, possibly at random), and exchange load information with it. After the exchange, the more heavily loaded processor can send processes to the other one until they are equally loaded. In this model, if 100 processes are suddenly created in an otherwise empty system, after one exchange we will have two machines with 50 processes, and after two exchanges most probably four machines with 25 processes. Processes diffuse around the network like a cloud of gas.

Actually migrating running processes is trivial in theory but close to impossible in practice. The hard part is not moving the code, data, and registers,

but moving the environment, such as the current position within all the open files, the current values of any running timers, pointers or file descriptors for communicating with tape drives or other I/O devices, etc. All of these problems relate to moving variables and data structures related to the process that are scattered about inside the operating system. What is feasible in practice is to use the load information to create new processes on lightly loaded machines, rather than trying to move running processes.

If one has adopted the idea of creating new processes only on lightly loaded machines, another approach, called bidding, is possible [FARBER and LARSON 1972; STANKOVIC and SIDHU 1984]. When a process wants some work done, it broadcasts a request for bids, telling what it needs (e.g., a 68000 CPU, 512K memory, floating point, and a tape drive).

Other processors can then bid for the work, telling what their workload is, how much memory they have available, etc. The process making the request then chooses the most suitable machine and creates the process there. If multiple request-for-bid messages are outstanding at the same time, a processor accepting a bid may discover that the workload on the bidding machine is not what it expected because that processor has bid for and won other work in the meantime.

3.3.4. Distributed Deadlock Detection

Some theoretical work has been done in the area of detection of deadlocks in distributed systems. How applicable this work may be in practice remains to be seen. Two kinds of potential deadlocks are **resource deadlocks** and **communication deadlocks**. Resource deadlocks are traditional deadlocks, in which some set of processes are all blocked waiting for resources held by other blocked processes. For example, if A holds X and B holds Y, and A wants Y and B wants X, a deadlock will result.

In principle, this problem is the same in centralized and distributed systems, but it is harder to detect in the latter because there are no centralized tables giving the status of all resources. The problem has mostly been studied in the context of data base systems [GLIGOR and SHATTUCK 1980; ISLOOR and MARSLAND 1978; MENASCE and MUNTZ 1979; OBERMARCK 1982].

The other kind of deadlock that can occur in a distributed system is a communication deadlock. Suppose A is waiting for a message from B and B is waiting for C and C is waiting for A. Then we have a deadlock. CHANDY et al. [1983] present an algorithm for detecting (but not preventing) communication deadlocks. Very crudely summarized, they assume that each process that is blocked waiting for a message knows which process or processes might send the message. When a process logically blocks, they assume that it does not really block, but instead sends a query message to each of the processes that might send it a real (data) message. If any of these processes is blocked, it sends query messages to the processes it is waiting for. If certain messages eventually come back to the original process, it can conclude that a deadlock exists. In effect, the algorithm is looking for a knot in a directed graph.

3.4. Fault Tolerance

Proponents of distributed systems often claim that such systems can be more reliable than centralized systems. Actually, there are at least two issues involved here: reliability and availability. Reliability has to do with the system not corrupting or losing your data. Availability has to do with the system being up when you need it. A system could be highly reliable in the sense that it never loses data, but at the same time be down most of the time and hence hardly usable. However, many people use the term "reliability" to cover availability as well, and we will not make the distinction either in the rest of the paper.

The reason why distributed systems are potentially more reliable than a centralized system is that if a system only has one instance of some critical component, such as a CPU, disk, or network interface, and that component fails, the system will go down. When there are multiple instances, the system may be able to continue in spite of occasional failures. In addition to hardware failures, one can also consider software failures. These are of two types: the software failed to meet the formal specification (implementation error), or the specification does not correctly model what the customer wanted (specification error). All work on program verification is aimed at the former, but the latter is also an issue. Distributed systems allow both hardware and software errors to be dealt with, albeit in somewhat different ways.

An important distinction should be made between systems that are fault tolerant and those that are fault intolerant. A fault tolerant system is one that can continue functioning (perhaps in a degraded form) even if something goes wrong. A fault intolerant system collapses as soon as any error occurs. Biological systems are highly fault tolerant; if you cut your finger, you probably will not die. If a memory failure garbles 1/10 of 1 percent of the program code or stack of a running program, the program will almost certainly crash instantly upon encountering the error.

It is sometimes useful to distinguish between expected faults and unexpected faults. When the ARPAnet was designed, people expected to lose packets from time to time. This particular error was expected and precautions were taken to deal with it. On the other hand, no one expected a memory error in one of the packet switching machines to cause that machine to tell the world that it had a delay time of zero to every machine in the network, which resulted in all network traffic being rerouted to the broken machine.

One of the key advantages of distributed systems is that there are enough resources to achieve fault tolerance, at least with respect to expected errors. The system can be made to tolerate both hardware and software errors, although it should be emphasized that in both cases it is the software, not the hardware, that cleans up the mess when an error occurs. In the past few years, two approaches to making distributed systems fault tolerant have emerged. They differ radically in orientation, goals, and attitude toward the theologically sensitive issue of the perfectability of mankind (programmers in particular). One approach is based on redundancy and the other is based on the notion of an atomic transaction. Both are described briefly below.

3.4.1. Redundancy Techniques

All the redundancy techniques that have emerged take advantage of the existence of multiple processors by duplicating critical processes on two or more machines. A particularly simple, but effective, technique is to provide every process with a backup process on a different processor. All processes communicate by message passing. Whenever anyone sends a message to a process, it also sends the same message to the backup process, as shown in Figure 8. The system ensures that neither the primary nor the backup can continue running until it has been verified that both have correctly received the message.

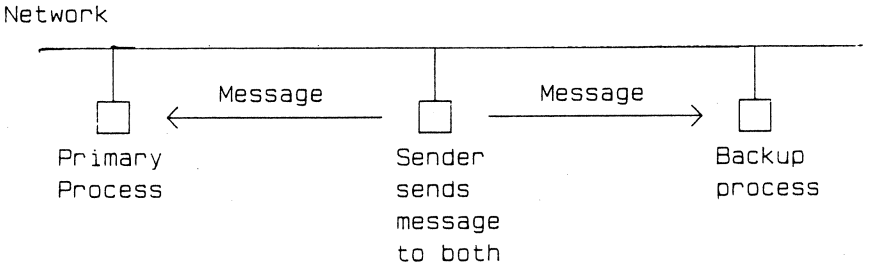


FIGURE 8. Each process has its own backup process.

Thus, if one process crashes due to any hardware fault, the other one can continue. Furthermore, the remaining process can then clone itself, making a new backup to maintain the fault tolerance in the future. BORG et al. [1983] have described a system using these principles.

One disadvantage of duplicating every process is the extra processors required, but another, more subtle problem, is that if processes exchange messages at a high rate, a considerable amount of CPU time may go into keeping the processes synchronized at each message exchange. POWELL and PRESOTTO [1983] have described a redundant system that puts almost no additional load on the processes being backed up. In their system, all messages sent on the network are recorded by a special "recorder" process (see Figure 9). From time to time, each process checkpoints itself onto a remote disk.

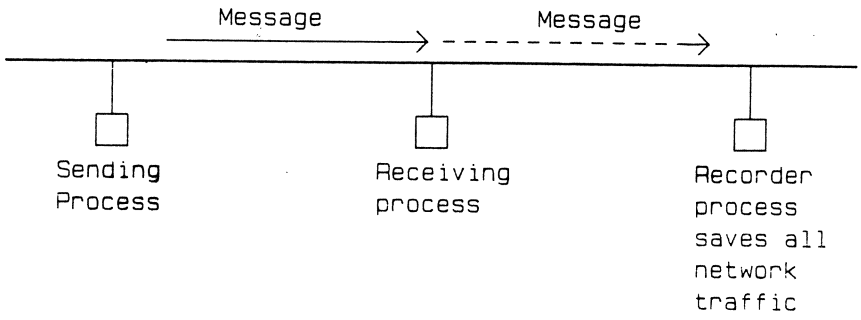


FIGURE 9. A recorder process copies and stores all network traffic without affecting the sender and receiver.

If a process crashes, recovery is done by sending the most recent checkpoint to an idle processor and telling it to start running. The recorder process then spoon feeds it all the messages that the original process received between the checkpoint and the crash. Messages sent by the newly restarted process are discarded. Once the new process has worked its way up to the point of crash, it begins sending and receiving messages normally, without help from the recording process.

The beauty of this scheme is that the only additional work a process must do to become immortal is to checkpoint itself from time to time. In theory, even the checkpoints can be disposed with, if the recorder process has enough disk space to store all the messages sent by all the currently running processes. If no checkpoints are made, when a process crashes, the recorder will have to replay the process's whole history.

When a process successfully terminates, the recorder no longer has to worry about having to rerun it, so all the messages that it received can be safely discarded. For servers and other processes that never terminate, this idea must be varied to avoid repeating individual transactions that have successfully completed.

One drawback of this scheme is that it relies on reliable reception of all messages all the time. In practice, local networks are very reliable, but they are not perfect. If occasional messages can be lost, the whole scheme becomes much less attractive.

Still, one has to be very careful about reliability, especially when the problem is caused by faulty software. Suppose a processor crashes due to a software bug. Both the schemes discussed above (Borg et al., and Powell and Presotto) deal with crashes by allocating a spare processor and restarting the crashed program, possibly from a checkpoint. Of course the new processor

will crash too, leading to the allocation of yet another processor and another crash. Manual intervention will eventually be required to figure out what is going on. If the hardware designers could provide a bit somewhere that tells whether a crash was due to hardware or software, it would be very helpful.

Both of the above techniques only apply to tolerance of hardware errors. However, it is also possible to use redundancy in distributed systems to make systems tolerant of software errors. One approach is to structure each program as a collection of modules, each one with a well-defined function and a precisely specified interface to the other modules. Instead of writing a module only once, N programmers are asked to program it, yielding N functionally identical modules.

During execution, the program runs on N machines in parallel. After each module finishes, the machines compare their results and vote on the answer. If a majority of the machines say that the answer is X, then all of them use X as the answer, and all continue in parallel with the next module. In this manner, the effects of an occasional software bug can be voted down. If formal specifications for any of the modules are available, the answers can also be checked against the specifications to guard against the possibility of accepting an answer that is clearly wrong.

A variation of this idea can be used to improve system performance. Instead of always waiting for all the processes to finish, as soon as k of them agree on an answer, those that have not yet finished are told to drop what they are doing, accept the value found by the k processes, and continue with the next module. Some work in this area is discussed in [AVIZIENIS and CHEN 1977; AVIZIENIS and KELLY 1984; ANDERSON and LEE 1981].

3.4.2. Atomic transactions

When multiple users on several machines are concurrently updating a distributed data base and one or more machines crash, the potential for chaos is truly impressive. In a certain sense, the current situation is a step backward from the technology of the 1950s, when the normal way of updating a data base was to have one magnetic tape, called the "master file," and one or more tapes with updates (e.g., daily sales reports from all of a company's stores). The master tape and updates were brought to the computer center, which then mounted the master tape and one update tape, and ran the update program to produce a new master tape. This new tape was then used as the "master" for use with the next update tape.

This scheme had the very real advantage that if the update program crashed, one could always fall back on the previous master tape and the update tapes. In other words, an update run could be viewed as either running correctly to completion (and producing a new master tape), or having no effect at all (crash part way through, new tape discarded). Furthermore, update jobs from different sources always ran in some (undefined) sequential order. It never happened that two users would concurrently read a field in a record, (e.g., 6), each add 1 to the value, and each store a 7 in that field, instead of the first one storing a 7 and the second storing an 8.

The property of run-to-completion or do-nothing is called an **atomic update**. The property of not interleaving two jobs is called **serializability**. The goal of people working on the atomic transaction approach to fault tolerance has been to regain the advantages of the old tape system without giving up the convenience of data bases on disk that can be modified in place, and to be able to do everything in a distributed way.

LAMPSON [1981] has described a way of achieving atomic transactions by building up a hierarchy of abstractions. We will summarize his model below. Real disks can crash during READ and WRITE operations in unpredictable ways. Furthermore, even if a disk block is correctly written, there is a small (but nonzero) probability of it subsequently being corrupted by newly developed bad spot on the disk surface. The model assumes that spontaneous block corruptions are sufficiently infrequent that the probability of *two* such events happening within some predetermined time, T, is negligible. To deal with real disks, the system software must be able to tell if a block is valid or not, for example, by using a checksum.

The first layer of abstraction on top of the real disk is the "careful disk," in which every CAREFUL-WRITE is read back immediately to verify that it is correct. If the CAREFUL-WRITE persistently fails, the system marks the block as "bad" and then intentionally crashes. Since CAREFUL-WRITEs are verified, CAREFUL-READs will always be good, unless a block has gone bad after being written and verified.

The next layer of abstraction is **stable storage**. A stable storage block consists of an ordered pair of careful blocks, which are typically corresponding careful blocks on different drives to minimize the chance of both being damaged by a hardware failure. The stable storage algorithm guarantees that at least one of the blocks is always valid. The STABLE-WRITE primitive first does a CAREFUL-WRITE on one block of the pair, and then the other. If the first one fails, a crash is forced, as mentioned above, and the second one is left untouched.

After every crash, and at least once every time period T, a special cleanup process is run to examine each stable block. If both blocks are "good" and identical, nothing has to be done. If one is "good" and one is "bad" (failure during a CAREFUL-WRITE), the "bad" one is replaced by the "good" one. If both are "good" but different (crash between two CAREFUL-WRITEs), the second one is replaced by a copy of the first one. This algorithm allows individual disk blocks to be updated atomically and survive infrequent crashes.

Stable storage can be used to create "stable processors" [Lampson 1981]. To make itself crashproof, a CPU must checkpoint itself on stable storage periodically. If it subsequently crashes, it can always restart itself from the last checkpoint. Stable storage can also be used to create stable monitors, in order to ensure that two concurrent processes never enter the same critical region at the same time, even if they are running on different machines.

Given a way to implement crashproof processors (stable processors) and crashproof disks (stable storage), it is possible to implement multicomputer atomic transactions. Before updating any part of the data in place, a stable

processor first writes an intentions list to stable storage, providing the new value for each datum to be changed. Then it sets a commit flag to indicate that the intentions list is complete. The commit flag is set by atomically updating a special block on stable storage. Finally it begins making all the changes called for in the intentions list. Crashes during this phase have no serious consequences because the intentions list is stored in stable storage. Furthermore, the actual making of the changes is idempotent, so repeated crashes and restarts during this phase are not harmful.

Atomic actions have been implemented in a number of systems; see for example [FRIDRICH and OLDER 1981; MITCHELL and DION 1982; BROWN et al. 1985; POPEK et al. 1981; REED and SVOBODOVA 1981].

3.5. SERVICES

In a distributed system, it is natural to provide functions by user-level server processes that have traditionally been provided by the operating system. This approach leads to a smaller (hence more reliable) kernel and makes it easier to provide, modify, and test new services. In the following sections, we will look at some of these services, but first we look at how services and servers can be structured.

3.5.1. Server structure

The simplest way to implement a service is to have one server that has a single, sequential thread of control. The main loop of the server looks something like this:

```
while true do
begin
    GetRequest;
    CarryOutRequest;
    SendReply
end
```

This approach is simple and easy to understand, but has the disadvantage that if the server must block while carrying out the request (e.g, in order to read a block from a remote disk), no other requests from other users can be started, even if they could have been satisfied immediately. An obvious example is a file server that maintains a large disk block cache, but occasionally must read from a remote disk. In the time interval that the server is blocked waiting for the remote disk to reply, it might have been able to service the next 10 requests, if they were all for blocks that happened to be in the cache. Instead, the time spent waiting for the remote disk is completely wasted.

To eliminate this wasted time and improve the throughput of the server, the server can maintain a table to keep track of the status of multiple partially completed requests. Whenever a request requires the server to send a message to some other machine and wait for the result, the server stores the status of the partially completed request in the table and goes back to the top of the

main loop to get the next message.

If the next message happens to be the reply from the other machine, that is fine and it is processed, but if it is a new request for service from a different client, that can also be started, and possibly completed before the reply for the first request comes in. In this way, the server is never idle if there is any work to be done.

Although this organization makes better use of the server's CPU, it makes the software much more complicated. Instead of doing nicely nested remote procedure calls to other machines whose services it needs, the server is back to using separate SEND and RECEIVE primitives, which are less structured.

One way of achieving both good performance and clean structure is to program the server as a collection of miniprocesses, which we will call a **cluster of tasks**. Tasks share the same code and global data, but each task has its own stack for local variables and registers and, most importantly, its own program counter. In other words, each task has its own thread of control. Multiprogramming of the tasks can be done either by the operating system kernel or by a run time library within each process.

There are two ways of organizing the tasks. The first way is to assign one task the job of "dispatcher," as shown in Figure 10. The dispatcher is the only task that accepts new requests for work. After inspecting an incoming request, it determines if the work can be done without blocking (e.g., if a block to be read is present in the cache). If it can, the dispatcher just carries out the work and sends the reply. If the work requires blocking, the dispatcher passes the work to some other task in the cluster, which can start work on it. When that task blocks, task switching occurs, and the dispatcher or some other previously blocked task can now run. Thus waiting for a remote procedure call to finish only blocks one task, not the whole server.

The other way of organizing the server is to have each task capable of accepting new requests for work. When a message arrives, the kernel gives it at random to one of the tasks listening to the address or port to which the message was addressed. That task carries the work out by itself, and no dispatcher is needed.

Both of these schemes require some method of locking the shared data to prevent races. This locking can be achieved explicitly by some kind of LOCK and UNLOCK primitives, or implicitly by having the scheduler not stop any task while it is running. For example, task switching only occurs when a task blocks. With ordinary user programs, such a strategy is undesirable, but with a server whose behavior is well-understood, it is not unreasonable.

3.5.2. File Service

There is little doubt that the most important service in any distributed system is the file service. Many file services and file servers have been designed and implemented, so a certain amount of experience is available [e.g., BIRRELL and NEEDHAM 1980; DELLAR 1982; DION 1980; FRIDRICH and OLDER 1981; FRIDRICH and OLDER 1984; MITCHELL and DION 1982; MULLENDER and

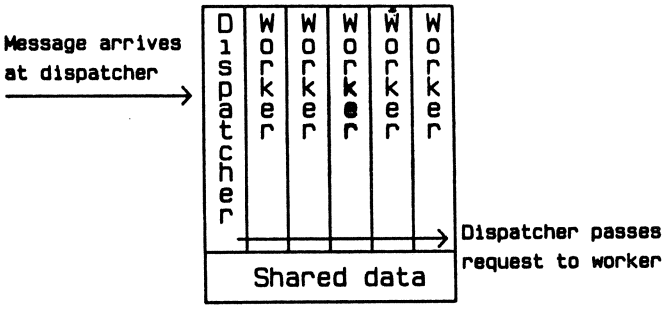


FIGURE 10. The dispatcher task waits for requests and passes them on to the worker tasks.

TANENBAUM 1985; REED and SVOBODOVA 1981; SATYANARAYANAN et al. 1985; SCHROEDER et al. 1985; STURGIS et al. 1980; SVOBODOVA 1981; SWINEHART et al. 1979]. A survey about file servers can be found in [SVOBODOVA 1984].

File services can be roughly classified into two kinds, "traditional" and "robust." Traditional file service is offered by nearly all centralized operating systems (e.g., the UNIX file system). Files can be opened, read, and rewritten in place. In particular, a program can open a file, seek to the middle of the file, and update blocks of data within the file. The file server implements these updates by simply overwriting the relevant disk blocks. Concurrency control, if there is any, usually involves locking entire files before updating them.

Robust file service, on the other hand, is aimed at those applications that require extremely high reliability, and whose users are prepared to pay a significant penalty in performance to achieve it. These file services generally offer atomic updates and similar features lacking in the traditional file service.

In the following paragraphs, we discuss some of the issues relating to traditional file service (and file servers) and then look at those issues that specifically relate to robust file service and servers. Since robust file service normally includes traditional file service as a subset, the issues covered in the first part also apply.

Conceptually, there are three components that a traditional file service normally has:

- Disk service
- Flat file service
- Directory service

The disk service is concerned with reading and writing raw disk blocks, without regard to how they are organized. A typical command to the disk service is to allocate and write a disk block, and return a capability or address (suitably protected) so the block can be read later.

The flat file service is concerned with providing its clients with an abstraction consisting of files, each of which is a linear sequence of records, possibly 1-byte records (as in UNIX) or client-defined records. The operations are reading and writing records, starting at some particular place in the file. The client need not be concerned with how or where the data in the file are stored.

The directory service provides a mechanism for naming and protecting files, so they can be accessed conveniently and safely. The directory service typically provides objects called directories that map ASCII names onto the internal identification used by the file service.

Design Issues. One important issue in a distributed system is how closely the three components of a traditional file service are integrated. At one extreme, the system can have distinct disk, file and directory services that run on different machines and only interact via the official interprocess communication mechanism. This approach is the most flexible, because anyone needing a different kind of file service (e.g., a B-tree file) can use the standard disk server. It is also potentially the least efficient, since it generates considerable interserver traffic.

At the other extreme, there are systems in which all three functions are handled by a single program, typically running on a machine to which a disk is attached. With this model, any application that needs a slightly different file naming scheme is forced to start all over making its own private disk server. However, the gain is increased runtime efficiency, because the disk, file and directory services do not have to communicate over the network.

Another important design issue in distributed systems is garbage collection. If the directory and file services are integrated, it is a straightforward matter to ensure that whenever a file is created, it is entered into a directory. If the directory system forms a rooted tree, it is always possible to reach every file from the root directory. However, if the file directory service and file service are distinct, it may be possible to create files and directories that are not reachable from the root directory. In some systems this may be acceptable, but in others, unconnected files may be regarded as garbage to be collected by the system.

Another approach to the garbage collection problem is to forget about rooted trees altogether, and permit the system to remove any file that has not been accessed for, say, 5 years. This approach is intended to deal with the situation of a client creating a temporary file and then crashing before recording its existence anywhere. When the client is rebooted, it creates a new temporary file and the existence of the old one is lost forever unless some kind of timeout mechanism is used.

There are a variety of other issues that the designers of a distributed file system must address; for example, will the file service be virtual-circuit oriented

or connectionless. In the virtual circuit approach, the client must do an OPEN on a file before reading it, at which time the file server fetches some information about the file (in UNIX terms, the i-node) into memory, and the client is given some kind of a connection identifier. This identifier is used in subsequent READs and WRITEs. In the connectionless approach, each READ request identifies the file and file position in full, so the server need not keep the i-node in memory (although most servers will maintain a cache for efficiency reasons).

Both virtual circuit and connectionless file servers can be used with the ISO OSI and RPC models. When virtual circuits are used for communication, having the file server maintain open files is natural. However, each request message can also be self contained, so that the file server need not hold the file open throughout the communication session.

Similarly, RPC fits well with a connectionless file server, but it can also be used with a file server that maintains open files. In the latter case, the client does an RPC to the file server to OPEN the file and get back a file identifier of some kind. Subsequent RPCs can do READ and WRITE operations using this file identifier.

The difference between these two becomes clear when one considers the effects of a server crash on active clients. If a virtual-circuit server crashes and is then quickly rebooted, it will almost always lose its internal tables. When the next request comes in to read the current block from file identifier 28, it will have no way of knowing what to do. The client will receive an error message, which will generally lead to the client process aborting. In the connectionless model, each request is completely self-contained (file name, file position, etc) so newly a reincarnated server will have no trouble carrying it out.

The price paid for this robustness, however, is a slightly longer message since each file request must contain the full file name and position. Furthermore, the virtual-circuit model is sometimes less complex in environments in which the network can re-order messages, that is, deliver the second message before the first one. Local networks do not have this defect, but some wide-area networks and internetworks do.

Protection. Another important issue faced by all file servers is access control—who is allowed to read and write which file. In centralized systems, the same problem exists, and is solved by using either an access control list or capabilities. With access control lists, each file is associated with a list of users who may access it. The UNIX RWX bits are a simple form of access control list that divides all users into 3 categories: owner, group, and others. With capabilities, a user must present a special "ticket" on each file access proving that he has access permission. Capabilities are normally maintained in the kernel to prevent forgery.

With a distributed system using remote file servers, both of these approaches have problems. With access control lists the file server has to verify that the user in fact is who he claims to be. With capabilities, how do you prevent users from making them up?

One way to make access control lists viable is to insist that the client first set up an authenticated virtual circuit with the file server. The authentication may involve a trusted third party as in [BIRRELL et al. 1982; BIRRELL et al. 1984]. When remote procedure calls are used, setting up an authenticated session in advance is less attractive. The problem of authentication using RPC is discussed in [BIRRELL 1985].

With capabilities, the protection is normally due to the fact that the kernel can be trusted. With personal computers on a network, how can the file server trust the kernel? After all, a user can easily boot up a nonstandard kernel on his machine. A possible solution is to encrypt the capabilities, as discussed in [MULLENDER and TANENBAUM 1984, 1985, 1986; TANENBAUM et al. 1986].

Performance. Performance is one of the key problems in using remote file servers (especially from diskless workstations). Reading a block from a local disk requires a disk access and a small amount of CPU processing. Reading from a remote server has the additional overhead of getting the data across the network. This overhead has two components, the actual time to move the bits over the wire (including contention resolution time, if any), and the CPU time the file server must spend running the protocol software.

CHERITON and ZWAENEPOEL [1983] describe measurements of network overhead in the context of the V system. With a 8 MHz 68000 processor and a 10 MB/sec Ethernet, they observe that reading a 512-byte block from the local machine takes 1.3 msec and from a remote machine 5.7 msec, assuming that the block is in memory and no disk access is needed. They also observe that loading a 64K program from a remote file server takes 255 msec vs. 60 msec locally, when transfers are in 16K units. A tentative conclusion is that access to a remote file server is four times as expensive as to a local one. (It is also worth noting that the V designers have gone to great lengths to achieve good performance; many other file servers are much slower than V's.)

One way to improve the performance of a distributed file system is to have both clients and servers maintain caches of disk blocks and possibly whole files. However, maintaining distributed caches has a number of serious problems. The worst of these is what happens when someone modifies the "master copy" on the disk? Does the file server tell all the machines maintaining caches to purge the modified block or file from their caches by sending them "unsolicited messages" as in XDFS [STURGIS, et al. 1980]? How does the server even know who has a cache? Introducing a complex centralized administration to keep track is probably not the way to go.

Furthermore, even if the server did know, having the server initiate contact with its clients is certainly an unpleasant reversal of the normal client-server relationship, in which clients make remote procedure calls on servers, but not vice versa. More research is needed in this area before we have a satisfactory solution. Some results are presented in [SCHROEDER et al. 1985].

Reliability. Reliability is another key design issue. The simplest approach is to design the system carefully, use good quality disks, and make occasional tape backups. If a disk ever gets completely wiped out due to hardware failure, all the work done since the last tape backup is lost. Although this mode of operation may seem scary at first, nearly all centralized computer systems work this way, and with a mean time between failure of 20,000 or more hours for disks these days, it works pretty well in practice.

For those applications that demand a higher level of reliability, some distributed systems have a more robust file service, as mentioned at the beginning of this section. The simplest approach is mirrored disks: every WRITE request is carried out in parallel on two disk drives. At every instant the two drives are identical, and either one can take over instantly for the other in the event of failure.

A refinement of this approach is to have the file server offer stable storage and atomic transactions, as discussed earlier. Systems offering this facility are described in [BROWN et al. 1985; DION 1980; MITCHELL and DION 1982; NEEDHAM and HERBERT 1982; REED and SVOBODOVA 1981; STURGIS et al. 1980; SVOBODOVA 1981]. A detailed comparison of a number of file servers offering sophisticated concurrency control and atomic update facilities is given by SVOBODOVA [1984]. We will just touch on a few of the basic concepts here.

At least four different kinds of files can be supported by a file server. *Ordinary* files consist of a sequence of disk blocks that may be updated in place, and which may be destroyed by disk or server crashes. *Recoverable* files have the property that groups of WRITE commands can be bracketed by BEGIN TRANSACTION and END TRANSACTION, and that a crash or abort midway leaves the file in its original state. *Robust* files are written on stable storage, and contain sufficient redundancy to survive disk crashes (generally two disks are used). Finally, *Multiversion* files consist of a sequence of versions, each of which is immutable. Changes are made to a file by creating a new version. Different file servers support various combinations of these.

All robust file servers need some mechanism for handling concurrent updates to a file or group of files. Many of them allow users to lock a file, page, or record to prevent conflicting writes. Locking introduces the problem of deadlocks, which can be dealt with using two-phase locking [ESWARAN et al 1976] or timestamps [REED 1983].

When the file system consists of multiple servers working in parallel, it becomes possible to enhance reliability by replicating some or all files over multiple servers. Reading also becomes easier because the workload can now be split over two servers, but writing is much harder because multiple copies must be updated simultaneously, or this effect simulated somehow.

One approach is to distribute the data, but keep some of the control information (semi) centralized. In LOCUS [POPEK et al. 1981; WALKER et al. 1983], for example, files can be replicated at many sites, but when a file is opened, the file server at one site examines the OPEN request, the number and status of the file's copies, and the state of the network. It then chooses one site to carry out the OPEN and the subsequent READs and WRITEs. The

other sites are brought up to date later.

3.5.3. Print Service

Compared to file service, on which a great deal of time and energy has been expended by a large number of people, the other services seem rather meager. Still, it is worth saying at least a little bit about a few of the more interesting ones.

Nearly all distributed systems have some kind of print service, to which clients can send files or file names or capabilities for files with instructions to print them on one of the available printers, possibly with some text justification or other formatting beforehand. In some cases, the whole file is sent to the print server in advance, and the server must buffer it. In other cases, only the file name or capability is sent, and the print server reads the file block by block as needed. The latter strategy eliminates the need for buffering (read: a disk) on the server side, but can cause problems if the file is modified after the print command is given but prior to the actual printing. Users generally prefer "call by value" rather than "call by reference" semantics for printers.

One way to achieve the "call by value" semantics is to have a printer spooler server. To print a file, the client process sends the file to the spooler. When the file has been copied to the spooler's directory, an acknowledgement is sent back to the client.

The actual print server is then implemented as a print client. Whenever the print client has nothing to print, it requests another file or block of a file from the print spooler, prints it, and then requests the next one. In this way the print spooler is a server to both the client and the printing device.

Printer service is discussed in [JANSON et al. 1983; and NEEDHAM and HERBERT 1982].

3.5.4. Process Service

Every distributed operating system needs some mechanism for creating new processes. At the lowest level, deep inside the system kernel, there must be a way of creating a new process from scratch. One way is to have a FORK call, as UNIX does, but other approaches are also possible. For example, in Amoeba, it is possible to ask the kernel to allocate chunks of memory of given sizes. The caller can then read and write these chunks, loading them with the text, data, and stack segments for a new process. Finally, the caller can give the filled-in segments back to the kernel and ask for a new process built up from these pieces. This scheme allows processes to be created remotely or locally, as desired.

At a higher level, it is frequently useful to have a process server that one can ask whether there is a Pascal, troff, or some other service, in the system. If there is, the request is forwarded to the relevant server. If not, it is the job of the process server to build a process somewhere and give it the request. After, say, a VLSI design rule checking server has been created and has done its work, it may or may not be a good idea to keep it in the machine where it was

created, depending on how much work (e.g., network traffic) is required to load it, and how often it is called. The process server could easily manage a server cache on a least recently used basis, so that servers for common applications are usually preloaded and ready to go. As special-purpose VLSI processors become available for compilers and other applications, the process server should be given the job of managing them in a way that is transparent to the system's users.

3.5.5. Terminal Service

How the terminals are tied to the system obviously depends to a large extent on the system architecture. If the system consists of a small number of minicomputers, each with a well-defined and stable user population, then each terminal can be hardwired to the computer its user normally logs on to. If, however, the system consists entirely of a pool of processors that are dynamically allocated as needed, it is better to connect all the terminals to one or more terminal servers that serve as concentrators.

The terminal servers can also provide such features as local echoing, intraline editing, and window management, if desired. Furthermore, the terminal server can also hide the idiosyncracies of the various terminals in use by mapping them all onto a standard virtual terminal. In this way, the rest of the software deals only with the virtual terminal characteristics and the terminal server takes care of the mappings to and from all the real terminals. The terminal server can also be used to support multiple windows per terminal, with each window acting as a virtual terminal.

3.5.6. Mail Service

Electronic mail is a popular application of computers these days. Practically every university computer science department in the Western world is on at least one international network for sending and receiving electronic mail. When a site consists of only one computer, keeping track of the mail is easy. However, when a site has dozens of computers spread over multiple local networks, users often want to be able to read their mail on any machine they happen to be logged on to. This desire gives rise to the need for a machine-independent mail service, rather like a print service that can be accessed system wide. ALMES et al. [1985] discuss how mail is handled in the Eden system.

3.5.7. Time Service

There are two ways to organize a time service. In the simplest way, clients can just ask the service what time it is. In the other way, the time service can broadcast the correct time periodically, to keep all the clocks on the other machines in sync. The time server can be equipped with a radio receiver tuned to WWV or some other transmitter that provides the exact time down to the microsecond.

Even with these two mechanisms, it is impossible to have all processes exactly synchronized. Consider what happens when a process requests the time-of-day from the time server. The request message comes in to the server,

and a reply is sent back immediately. That reply must propagate back to the requesting process, cause an interrupt on its machine, have the kernel started up, and finally have the time recorded somewhere. Each of these steps introduces an unknown, variable delay.

On an Ethernet, for example, the amount of time required for the time server to put the reply message onto the network is nondeterministic and depends on the number of machines contending for access at that instant. If a large distributed system has only one time server, messages to and from it may have to travel a long distance and pass over store-and-forward gateways with variable queueing delays. If there are multiple time servers, they may get out of synchronization because their crystals run at slightly different rates. Einstein's special theory of relativity also puts constraints on synchronizing remote clocks.

The result of all these problems is that having a single, global time is impossible. Distributed algorithms that depend on being able to find a unique global ordering of widely separated events may not work as expected. A number of researchers have tried to find solutions to the various problems caused by the lack of global time. See for example [JEFFERSON 1985; LAMPORT 1984; LAMPORT 1978; MARZULLO and OWICKI 1985; REED 1983; REIF and SPIRAKIS 1984;]

3.5.8. Boot Service

The boot service has two functions: bringing up the system from scratch when the power is turned on, and helping important services survive crashes. In both cases, it is helpful if the boot server has a hardware mechanism for forcing a recalcitrant machine to jump to a program in its own ROM, in order to reset it. The ROM program could simply sit in a loop waiting for a message from the boot service. The message would then be loaded into that machine's memory and executed as a program.

The second function alluded to above is the "immortality service." An important service could register with the boot service, which would then poll it periodically to see if it were still functioning. If not, the boot service could initiate measures to patch things up, for example, forcibly reboot it or allocate another processor to take over its work. To provide high reliability, the boot service should itself consist of multiple processors, each of which keeps checking that the other ones are still working properly.

3.5.9. Gateway Service

If the distributed system in question needs to communicate with other systems at remote sites, it may need a gateway server to convert messages and protocols from internal format to those demanded by the wide-area network carrier.

4. Examples of Distributed Operating Systems

Having disposed with the principles, it is now time to look at some actual distributed systems that have been constructed as research projects in universities around the world. Although many such projects are in various stages of development, space limitations prevent us from describing all of them in detail. Instead of saying a few words about each system, we have chosen to look at four systems that we consider representative. Our selection criteria were as follows. First, we only chose systems that were designed from scratch as distributed systems, (systems that gradually evolved by connecting together existing centralized systems or are multiprocessor versions of UNIX were excluded). Second, we only chose systems that have actually been implemented; paper designs did not count. Third, we only chose systems about which a reasonable amount of information was available.

Even with these criteria, there were many more systems that could have been discussed. As an aid to the reader interested in pursuing this subject further, we provide here some references to other relevant work: Accent [Fitzgerald and Rashid 1985; Rashid and Robertson 1981], Argus [Liskov 1982; Liskov 1984; Liskov and Scheifler 1982; Oki et al. 1985], Chorus [Zimmerman et al. 1981], CRYSTAL [DeWitt et al. 1984], DEMOS [Powell and Miller 1983], Distributed UNIX [Luderer et al. 1981], HXDP [Jensen 1978], LOCUS [Popek et al. 1981; Walker et al. 1983; Weinstein et al. 1985], Meglos [Gaglianello and Katseff 1985], MICROS [Curtis and Wittie 1984; Mohan and Wittie 1985; Wittie and Curtis 1985; Wittie and van Tilborg 1980], RIG [Ball et al. 1976], Roscoe/Arachne [Finkel et al. 1979; Solomon and Finkel 1979; Solomon and Finkel 1978], and the work at Xerox PARC [Birrell et al. 1984; Birrell and Nelson 1984; Birrell 1985; Boggs et al. 1980; Brown et al. 1985; Swinehart et al. 1979].

The systems we will examine here are: The Cambridge Distributed Computing System, Amoeba, V, and Eden. The discussion of each system follows the list of topics treated above, namely communication primitives, naming and protection, resource management, fault tolerance, and services.

4.1. The Cambridge Distributed Computing System

The Computing Laboratory at the University of Cambridge has been doing research in networks and distributed systems since the mid 1970s, first with the Cambridge ring and later with the Cambridge Distributed Computing System [Needham and Herbert 1982]. The Cambridge ring is not a token-passing ring, but rather contains several minipacket slots circulating around the ring. To send a packet, a machine waits until an empty slot passes by, then inserts a minipacket containing the source, destination, some flag bits, and 2 bytes of data. Although the 2-byte minipackets themselves are occasionally useful (e.g., for acknowledgements), several block-oriented protocols have been developed for reliably exchanging 2K packets by accumulating 1024 minipackets. The nominal ring bandwidth is 10 Mbps, but since each minipacket has 2 bytes of data and 3 bytes of overhead, the effective bandwidth is 4 Mbps.

The Cambridge ring project was very successful, with copies of the ring

currently in operation at many universities and companies in the U.K. and elsewhere. The availability of the ring led to research on distributed computing systems initially using nine Computer Automation LSI4 minicomputers and later using about a dozen Motorola 68000s, under the direction of Roger Needham.

The Cambridge system is primarily composed of two components: the processor bank and the servers. When a user logs in, he normally requests one machine from the processor bank, uses it as a personal computer for the entire work session, and returns it when logging out. Processors are not normally dynamically allocated for short periods of time. The servers are dedicated machines that provide various useful services, including file service, name service, boot service, etc. The number and location of these servers is relatively static.

4.1.1. Communication Primitives

Due to the evolution from network to distributed system described earlier, the communication primitives are usually described as network protocols rather than as language primitives. The choice of the primitives was closely tuned to the capabilities of the ring in order to optimize performance. Nearly all communication is built up from sending packets consisting of a 2-byte header, a 2-byte process identifier, up to 2048 data bytes, and a 2-byte checksum. On top of this basic packet protocol are a simple remote procedure call protocol and a byte stream protocol.

The basic packet protocol, which is a pure datagram system, is used by the **single shot** protocol to build up something similar to a remote procedure call. It consists of having the client send a packet to the server containing the request, and the having the server send a reply. Some machines are multiprogrammed, so the second minipacket (called 'route' above) is used to route the incoming packet to the correct process. The request packet itself contains a function code and the parameters, if any. The reply packet contains a status code and the result, if any. Clients do not acknowledge receipt of the result.

Some applications, such as terminal handling and file transfer work better with a flow-controlled, virtual circuit protocol. The **byte stream protocol** is used for these applications. This protocol is a full-duplex connection-oriented protocol, with full flow control and error control.

4.1.2. Naming and Protection

Services can be located in the Cambridge system by using the name server. To look up a name, the client sends an ASCII string to the name server, which then looks it up in its tables and returns the machine number where the service is located, the port used to address it, and the protocol it expects. The name server stores service names as unstructured ASCII strings, which are simply matched against incoming requests character by character, that is, it does not manage hierarchical names. The name server itself has a fixed address that never changes, so this address may be embedded into programs.

Although the service data base is relatively static, from time to time names

must be added or deleted to the name server's data base. Commands are provided for this purpose, but for protection reasons these commands may only be executed by the system administrator.

Finding the location of a service is only half the work. To use most services, a process must identify itself in an unforgeable way, so the service can check to see if that user is authorized. This identification is handled by the Active Name Server, which maintains a table of currently logged in users. Each table entry has four fields: the user's login name, his session key (a big random number), the user's class (e.g., faculty, student) and a control key, as shown in Figure 11.

Login	Session	Class	Control
MARVIN	91432	STUDENT	31513
BARBARA	61300	STUDENT	27138
ANDY	42108	FACULTY	31618
SUZANNE	81346	DIRECTOR	41948

FIGURE 11. The Active Name Table.

To use a service, a user supplies the service with his login name, session key (obtained at login time), and class. The service can then ask the Active Name Server if such an entry exists. Since session keys are sparse, it is highly unlikely that a student will be able to guess the current session key for the computer center director, and thus be able to obtain services reserved for the director. The control key must be presented to change an entry, thus providing a mechanism to restrict changing the Active Name Server's table to a few people.

4.1.3. Resource Management

The main resource managed by the system is the processor bank, handled by a service called the **resource manager**. Usually a user requests a processor to be allocated at login time, and then loads it with a single-user operating system. The processor then becomes the user's personal computer for the rest of the login session.

The resource manager accepts requests to allocate a processor. In these requests, the user specifies a CPU type (e.g., 68000), a list of attributes (e.g. memory size), and a program to be run. The resource manager then selects the most suitable CPU currently available to allocate. Various defaults are available, so, for example, a user can specify that he wants to run TRIPOS (a straightforward single-user operating system), and the resource manager will select an appropriate CPU type if none has been specified.

The downloading of programs into processor bank machines is controlled by a server called the **ancilla**, although some of the machines have intelligent ring interfaces that actually do most of the work. The ancilla also helps simulate the machine's console and front panel, so users have the same control over a processor bank machine as they would over a real personal computer on their desks.

4.1.4. Fault Tolerance

The approach taken to fault tolerance in the Cambridge system is to make it easy to bring servers back up after a crash. When a ring interface detects a special minipacket whose source is the name server, it reboots the processor by forcing it to jump to a program in ROM. This program then sends a request to the boot server, which in turn goes to the name server asking for reverse name lookup. The name server then searches its tables to find the service that is running on the machine from which the reverse lookup request came. As soon as the reply comes in, the server knows what it is supposed to be doing, and can request the resource manager and ancilla to download the appropriate program. When machines are physically reset or powered up, the same procedure is carried out automatically.

Another area in which some effort has been put to make the system fault tolerant is the file system, which supports atomic updates on special files. This facility is described in the next section.

4.1.5. Services

We have already described several key servers, including the name server, resource manager, ancilla, and active name server. Other small servers include the time server, print server, login server, terminal server, and error server, which records system errors for maintenance purposes. The file server is examined here.

The file system started out with the idea of a single universal file server that provided basic storage service but very primitive naming and protection system, coupled with single-user TRIPOS operating systems in the processor bank machines, in which the naming and directory management would be done. The CAP computer (a large research machine within the Laboratory that does not ha ve any disks of its own) also uses the file server. After some experience with this model, it was decided to create a new server, known as the **filing machine**, as a front end to the file system to improve the performance (mostly by providing the filing machine with a large cache, something that the small user machines could not afford). The CAP machine, which has adequate memory, continues to use the file server directly. The position of the filing machine is shown in Figure 12.

The universal file server supports one basic file type, with two minor variations. The basic file type is an unstructured file consisting of a sequence of 16-bit words, numbered from 0 to some maximum. Operations are provided for reading or writing arbitrary numbers of words, starting anywhere in the

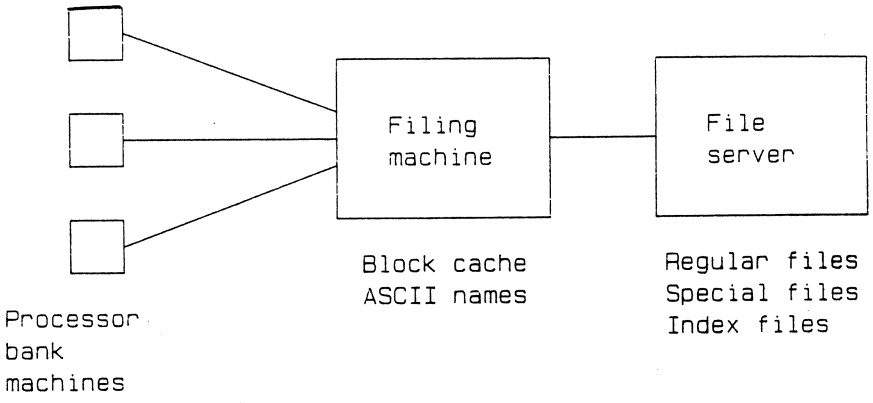


FIGURE 12. The filing machine is positioned between the users and the file server. It maintains a block cache and handles ASCII names.

file. Each file is uniquely identified by a 64-bit PUID (Permanent User IDentifier) consisting of a 32-bit disk address and a 32-bit random number.

The first variation is the special file, which has the property that writes to it are atomic, that is, they will either succeed completely or not be done at all. They will never be partly completed, even in the face of server crashes.

The second variation is a file called an **index**, which is a special file consisting of a sequence of slots, each holding one PUID. When a file is created, the process creating it must specify an index and slot in that index into which the new file's PUID is stored. Since indices are also files, and as such have PUIDs themselves, an index may contain pointers (PUIDs) to other indices, allowing arbitrary directory trees and graphs to be built. One index is distinguished as being the root index, which has the property that the file server's internal garbage collector will never remove a file reachable from the root index.

In the initial implementation, the full code of the TRIPOS operating system was loaded into each pool processor. All of the directory management and handling of ASCII names was done on the processor bank machines. Unfortunately, this scheme had several problems. First, TRIPOS was rather large and filled up so much memory that little room was left for buffers, meaning that almost every read or write request actually caused a disk access (the universal file server has hardly any buffers). Second, looking up a name in the directory hierarchy required all the intermediate directories between the starting point and the file to be physically transported from the file server to a

machine doing the search.

To get around these problems, a filing machine with a large cache was inserted in front of the file server. This improvement allowed programs to request files by name instead of PUID, with the name look up occurring in the filing machine now. Due to the large cache, most of the relevant directories are likely to already be present in the filing machine, thus eliminating much network traffic. Furthermore, it allowed the TRIPOS code in the user machines to be considerably stripped, since the directory management was no longer needed. It also allowed the file server to read and write in large blocks; this was previously possible, but rarely done due to lack of buffer space on the user side. The resulting improvements were substantial.

4.1.6. Implementation

As should be clear by now, the whole Cambridge system is a highly pragmatic design, which from its inception [WILKES and NEEDHAM 1980] was designed to actually be used by a substantial user community. About 90 machines are connected by three rings now, and the system is fairly stable. A related research project was the connection of a number of Cambridge rings via a satellite [ADAMS et al. 1982]. Future research may include interconnection of multiple Cambridge rings using very high speed (2 Mbit/sec) lines.

4.2. AMOEBA

Amoeba is a research project on distributed operating systems being carried out at the Vrije Universiteit in Amsterdam under the direction of Andrew Tanenbaum. Its goal is to investigate capability-based, object-oriented systems, and to build a working prototype system to use and evaluate. It currently runs on a collection of 24 Motorola 68010 computers connected by a 10 Mbps local network.

The Amoeba architecture consists of four principal components, as shown in Figure 13. First are the workstations, one per user, on which users can carry out editing and other tasks that require fast interactive response. Second are the pool processors, a group of CPUs that can be dynamically allocated as needed, used, and then returned to the pool. For example, the "make" command might need to do six compilations, so six processors could be taken out of the pool for the time necessary to do the compilation and then returned. Alternatively, with a five-pass compiler, $5 \times 6 = 30$ processors could be allocated for the six compilations, gaining even more speedup.

Third are the specialized servers, such as directory, file, and block servers, data base servers, bank servers, boot servers, and various other servers with specialized functions. Fourth are the gateways, which are used to link Amoeba systems at different sites (and, eventually, different countries) into a single, uniform system.

All the Amoeba machines run the same kernel, which primarily provides message-passing services and little else. The basic idea behind the kernel was to keep it small, not only to enhance its reliability, but also to allow as much

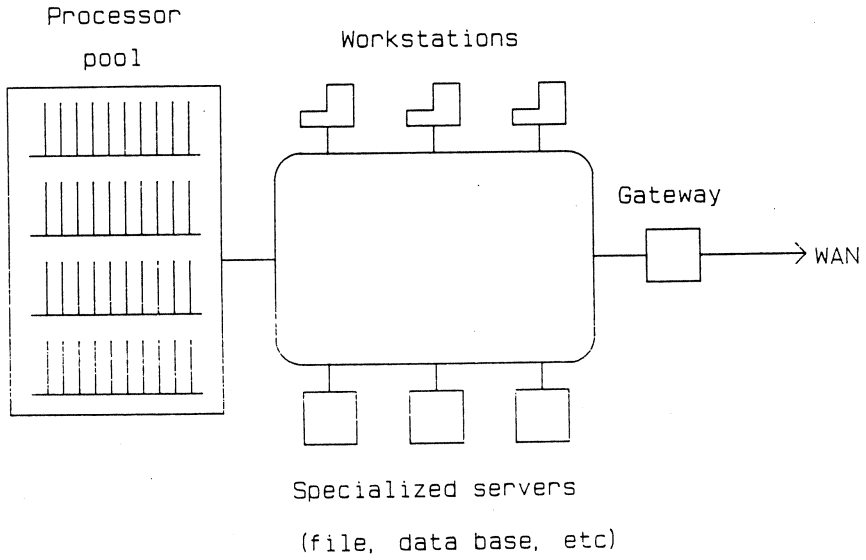


FIGURE 13. The Amoeba architecture.

as possible of the operating system to run as user processes, providing for flexibility and experimentation.

Some of the research issues addressed by the project are how to put as much of the operating system as possible into user processes, how to use the processor pool, how to integrate the workstations and processor pool, and how to connect multiple Amoeba sites into a single coherent system using wide-area networks. All of these issues use objects and capabilities in a uniform way.

4.2.1. Communication Primitives

The conceptual model for Amoeba communication is the abstract data type or object model, in which clients perform operations on objects in a location independent manner. To implement this model, Amoeba uses a minimal remote procedure call model for communication between clients and servers. The basic client primitive is to send a message of up to about 32K bytes to a server and then block waiting for the result. Servers use GET_REQUEST and PUTREPLY to get new work and send back the results, respectively. These primitives are not embedded in a language environment with automatic stub generation. They are implemented as small library routines that are used to

invoke the kernel directly from C programs.

All the primitives are reliable in the sense that detection and retransmission of lost messages, acknowledgement processing, and message-to-packet and packet-to-message management are all done transparently by the kernel. Messages are unbuffered. If a message arrives and no one is expecting it, the message is simply discarded. The sending kernel then times out and tries again. Users can specify how long the kernel should retransmit before giving up and reporting failure. The idea behind this strategy is that server processes are generally cloned in *N*-fold, so normally there will be a server waiting. Since a message is discarded only if the system is badly overloaded, having the client time out and try again later is not a bad idea.

Although the basic message primitives are blocking, special provision is made for handling emergency messages. For example, if a data base server is currently blocked waiting for a file server to get some data for it, and a user at a terminal hits the BREAK key (indicating that he wants to kill off the whole request), some way is needed to gracefully abort all the processes working on behalf of that request. In the Amoeba system the terminal server generates and sends a special EXCEPTION message, which causes an interrupt at the receiving process.

This message forces the receiver to stop working on the request and send an immediate reply with a status code of REQUEST ABORTED. If the receiver was also blocked waiting for a server, the exception is recursively propagated all the way down the line, forcing each server in turn to finish immediately. In this manner, all the nested processes terminate normally (with error status), so little violence is done to the nesting structure. In effect, an EXCEPTION message does not terminate execution. Instead, it just says "Force normal termination immediately, even if you are not done yet, and return an error status."

4.2.2. Naming and Protection

All naming and protection issues in Amoeba are dealt with by a single, uniform mechanism: sparse capabilities [TANENBAUM et al. 1986]. The system supports objects such as directories, files, disk blocks, processes, bank accounts, devices, etc., but not small objects such as integers. Each object is owned by some service and managed by the corresponding server processes.

When an object is created, the process requesting its creation is given a capability for it. Using this capability, a process can carry out operations on the object, such as reading or writing the blocks of a file, starting or stopping a process etc. The number and type of operations applicable to an object are determined by the service that created the object; a bit map in the capability tells which of those the holder of the capability is permitted to use. Thus the whole of Amoeba is based on the conceptual model of abstract data types managed by services, as mentioned above. Users view the Amoeba environment as a collection of objects, named by capabilities, on which they can perform operations. This is in contrast to systems where the user view is a collection of processes connected by virtual circuits.

Each object has a globally unique name, contained in its capabilities.

Capabilities are managed entirely by user processes; they are protected cryptographically, not by any kernel maintained tables or mechanisms. A capability has four fields as shown in Figure 14:

1 The *service port*: a sparse address corresponding to the service that owns the object, such as a file or directory service.
2 The *object number*: an internal identifier that the service uses to tell which of its objects this is (cf. the i-number in UNIX).
3 The *rights field*: a bit map telling which operations on the object are permitted.
4 The *check field*: a large random number used to authenticate the capability.

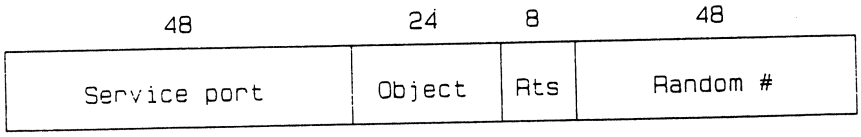


FIGURE 14. An Amoeba capability.

When a server is asked to create an object, it picks an available slot in its internal tables (e.g., a free i-node, in UNIX terminology), puts the information about the new object there, and picks a new random number to be used exclusively to protect this new object. Each server is free to use any protection scheme it wants to, but the normal one is for it to build a capability containing its port, the object number, the rights (initially all present), and a known constant. The two latter fields are then thoroughly mixed by encrypting them with the random number as key, which is then stored in the internal table.

Later, when a process performs an operation on the object, a message containing the object's capability is sent to the server. The server uses the (plaintext) *object number* to find the relevant internal table entry and extract the random number, which is then used to decrypt the *rights* and *check* fields. If the decryption yields the correct known constant, the *rights* field is believed and the server can easily check if the requested operation is permitted. More details about protection of capabilities can be found in [MULLENDER and TANENBAUM 1986; MULLENDER and TANENBAUM 1984; TANENBAUM et al. 1986].

Capabilities can be stored in directories managed by the directory service. A directory is effectively a set of (ASCII string, capability) pairs. The most common directory operation is for a user to present the directory server with a capability for a directory (itself an object) and an ASCII string and ask for the capability that corresponds to that string in the given directory. Other operations are entering and deleting (ASCII string, capability) pairs.

This naming scheme is flexible in that a directory may contain capabilities for an arbitrary mixture of object types and locations, but it is also uniform in that every object is controlled by a capability. A directory entry may, of course, be for another directory, so it is simple to build up a hierarchical (e.g., UNIX-like) directory tree, or even more general naming graphs. Furthermore, a directory may also contain a capability for a directory managed by a different directory service. As long as all the directory services have the same interfaces with the user, one can distribute objects over directory services in an arbitrary way.

4.2.3. Resource Management

Resource management in Amoeba is performed in a distributed way, again using capabilities. Each Amoeba machine (pool processor, work station, etc.) runs a resource manager process that controls that machine. This process actually runs inside the kernel, for efficiency reasons, but it uses the normal abstract data type interface with its clients. The key operations it supports are CREATE SEGMENT, WRITE SEGMENT, READ SEGMENT, and MAKE PROCESS. To create a new process, a process would normally execute CREATE SEGMENT three times for the child process' text, data, and stack segments, getting back one capability for each segment. Then it would fill each one in with that segment's initial data, and finally perform MAKE PROCESS with these capabilities as parameters, getting back a capability for the new process.

Using the above primitives, it is easy to build a set of processes that share text and/or data segments. This facility is useful for constructing servers that consist internally of multiple miniprocesses (tasks) that share text and data. Each of these processes has its own stack, and most importantly, its own program counter, so that when one of them blocks on a remote procedure call, the others are not affected. For example, the file server might consist of 10 processes sharing a disk cache, all of which start out by doing a GET_REQUEST. When a message comes in, the kernel sees that 10 processes are all listening to the port specified in the message, so it picks one process at random and gives it the message. This process then performs the requested operation, possibly blocking on remote procedure calls (e.g., calling the disk) while doing so, but leaving the other server processes free to accept and handle new requests.

At a higher level, the processor pool is managed by a process server that keeps track of which ones are free and which ones are not. If an installation wants to multiprogram the processor pool machines, then the process server manages each process table slot on a pool processor as a virtual processor. One of the interesting research issues here is the interplay between the workstations and the processor pool, that is, when should a process be started up on the workstation and when should it be offloaded to a pool processor. Research has not yet yielded any definitive answers here, although it seems intuitively clear that highly interactive processes, such as screen editors, should be local to the workstation, and batch-like jobs, such as big compilations (e.g., UNIX

"make"), should be run elsewhere.

Accounting. Amoeba provides a general mechanism for resource management and accounting in the form of the **bank server**, which manages "bank account" objects. Bank accounts hold virtual money, possibly in multiple currencies. The principal operation on bank account objects is transferring virtual money between accounts. For example, to pay for file storage, a file server might insist on payment in advance of X dollars per megabyte of storage, and a phototypesetter server might want a payment in advance of Y yen per page. The system management can decide whether or not dollars and zlotys are convertible, depending on whether or not it wants users to have separate quotas on disk space and typesetter pages, or just give each user a single budget to use as he sees fit.

The bank server provides a basic mechanism on top of which many interesting policies can be implemented. For example, if some resource is in short supply, are servers allowed to raise the price as a rationing mechanism? Do you get your money back when you release disk space; that is, is the model one of clients and servers buying and selling blocks, or is it like renting something? If it is like renting, there will be a flow of money from users to the various servers, so users need incomes to keep them going, rather than simply initial fixed budgets. When new users are added, virtual money has to be created for them. Does this lead to inflation? The possibilities here are legion.

4.2.4. Fault Tolerance

The basic idea behind fault tolerance in Amoeba is that machine crashes are infrequent, and that most users are not willing to pay a penalty in performance in order to make all crashes 100% transparent. Instead, Amoeba provides a boot service, with which servers can register. The boot service polls each registered server at agreed upon intervals. If the server does not reply properly within a specified time, the boot service declares the server to be broken, and requests the process server to start up a new copy of the server on one of the pool processors.

To understand how this strategy affects clients, it is important to realize that Amoeba does not have any notion of a virtual circuit or a session. Each remote procedure call is completely self-contained and does not depend on any previous set up, that is, it does not depend on any volatile information stored in server's memories. If a server crashes before sending a reply, the kernel on the client side will time out and try again. When the new server comes up, the client's kernel will discover this and send the request there, without the client even knowing anything has happened. Of course, this approach does not always work, for example, if the request is not idempotent (the chocolate factory!) or if a sick disk head has just mechanically scraped all the bits from some disk surface, but it works much of the time and has zero overhead under normal conditions.

4.2.5. Services

Amoeba has several kinds of block, file and directory service. The simplest one is a server running on top of the Amoeba kernel that provides a file service functionally equivalent to the UNIX system call interface, to allow most UNIX programs to run on Amoeba with only the need to re-link them with a special library.

A more interesting server, however, is FUSS (Free University Storage System) which views each file as a sequence of versions. A process can acquire a capability for a private copy of a new version, modify it, and then commit it in a single indivisible atomic action. Providing atomic commits at the file level (rather than only as a facility in some data base systems), simplifies the construction of various servers, such as the bank server, that have to be highly robust. FUSS also supports multiple, simultaneous access using optimistic concurrency control. It is described in more detail in MULLENDER and TANENBAUM [1985].

Other key services are the directory service, bank service, and boot service, all of which have already been discussed.

4.2.6. Implementation

The Amoeba kernel has been ported to five different CPUs: 68010, NS32016, 8088, VAX, and PDP-11. version. All the servers described above, except the boot server, have been written and tested, along with a number of others. Measurements have shown that a remote procedure call from user space on one 68010 to user space on a different 68010 takes just over 8 msec (plus the time to actually carry out the service requested). The data rate between user processes on different machines has been clocked at over 250,000 bytes/sec, which is about 20% of the raw network bandwidth, an exceptionally high value.

A library has been written to allow UNIX programs to run on Amoeba. A substantial number of utilities, including compilers, editors, and shells are operational. A server has also been implemented on UNIX to allow Amoeba programs to put capabilities for UNIX files into their directories and use them without having to know that the files are actually located on a VAX running UNIX.

In addition to the UNIX emulation work, various applications have been implemented using pure Amoeba, including parallel traveling salesman and parallel alpha-beta search [BAL et al. 1985]. Current research includes connecting Amoeba systems at five locations in three countries using wide-area networks.

4.3. THE V KERNEL

The V kernel is a research project on distributed systems at Stanford University under the direction of David Cheriton [CHERITON 1984; CHERITON and ZWAENEPOEL 1984a; CHERITON and ZWAENEPOEL 1984b; CHERITON and MANN 1984]. It was motivated by the increasing availability of powerful microcomputer-based workstations, which can be seen as an alternative to

traditional time-shared minicomputers. The V kernel is an outgrowth of the experience acquired with earlier systems, Thoth [CHERITON 1982; CHERITON et al. 1979] and VEREX.

The V kernel can be thought of as a software backplane, analogous to the Multibus or S-100 bus backplanes. The function of a backplane is to provide an infrastructure for components (for hardware, boards; for software processes) to communicate, and nothing else. Consequently, most of the facilities found in traditional operating systems, such as a file system, resource management, and protection are provided in V by servers outside the kernel. In this respect V and Amoeba are conceptually very similar.

Another point on which V and Amoeba agree is the free market model of services. Services such as the file system are, in principle, just ordinary user processes. Any user who is dissatisfied with the standard file system [STONEBRAKER, 1981; TANENBAUM and MULLENDER 1982] is free to write his own. This view is in contrast to the "centrally planned economy" model of most timesharing systems, which present the file system on a "like it or lump it" basis.

The V system consists of a collection of workstations (currently SUNs) each running an identical copy of the V kernel. The kernel consists of three components: the interprocess communication handler, the kernel server (for providing basic services, such as memory management), and the device server (for providing uniform access to I/O devices). Some of the workstations support an interactive user, whereas others function as file servers, print servers, and other kinds of servers, as shown in Figure 15. Unlike Amoeba, V does not have a processor pool.

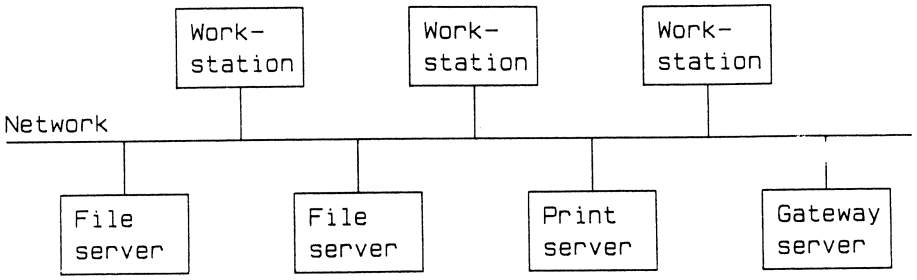


FIGURE 15. A typical V configuration.

4.3.1. Communication Primitives

The V communication primitives have been designed in accordance with the backplane model mentioned above. They provide basic, but fast communication. To access a server, a client does SEND(message, pid), which transmits the fixed-length (32-byte) 'message' to the server, and then blocks until the server has sent back a reply, which overwrites 'message.' The second parameter, 'pid,' is a 32-bit integer that uniquely identifies the destination process. A message may contain a kind of pseudo-pointer to one of the client's memory segments. This pseudo-pointer can be used to permit the server to read from or write to the client's memory. Such reads and writes are handled by kernel primitives COPYFROM and COPYTO. As an optimization, when a client does a SEND containing one of these pseudo-pointers with READ permission, the first 1K of the segment is piggybacked onto the message, on the assumption that the server will probably want to read it eventually. In this way, messages longer than 32 bytes can be achieved.

Servers use the RECEIVE and REPLY calls. The RECEIVE call can provide a segment buffer in addition to the regular message buffer, so that if (part of) a segment was piggybacked onto the message, it will have a place to go. The REPLY call can also provide a segment buffer, for the case that the client provided a pseudo-pointer that the server could use to return results exceeding 32 bytes.

To make this communication system easier to use, calls to servers can be embedded in stubs so the caller just sees an ordinary procedure call. Stub generation is not automated, however.

4.3.2. Naming and Protection

V has three levels of naming. At the bottom level, each process has a unique 32-bit pid, which is the address used to send messages to it. At the next level, services (i.e, processes that carry out requests for clients) can have symbolic (ASCII string) names in addition to their pids. A service can register a symbolic name with its kernel so that clients can use the symbolic name instead of the pid. When a client wants to access a service by its name, the client's kernel broadcasts a query to all the other kernels, to see where the server is. The (ServerName, pid) pair is then put in a cache for future use.

The top level of naming makes it possible to assign symbolic names to objects, such as files. Symbolic names are always interpreted in some "context," analogous to looking up a file name in some directory in other systems. A context is a set of records, each including the symbolic name, server's pid, context number and object identifier. Each server manages its own contexts; there is no centralized "name server." A symbolic name is looked up in a context by searching all the records in that context for one whose name matches the given name. When a match is found, the context number and object identifier can be sent to the appropriate server to have some operation carried out.

Names may be hierarchical, as in a/b/c. When "a" is looked up in some context, the result will probably be a new context, possibly managed by a new

server on a different machine. In that case the remaining string, "b/c" is passed on to that new server for further lookup, and so on.

It is also possible to prefix a symbolic name with an explicit context, as in "[HomeDirectory] a/b/c", in which case the name is looked up in the context specified, rather than in the current context (analogous to the current working directory in other systems). A question that quickly arises is, "Who keeps track of the various context names, such as "HomeDirectory" above?" The answer is that each workstation in the system has a Context Prefix Server, whose function is to map context names onto server names, so that the appropriate server can be found to interpret the name itself.

4.3.3. Resource Management

Each processor in V has a dedicated function, either as a user workstation or a file, print, or other dedicated server, so no form of dynamic processor allocation is provided. The key resources to be managed are processes, memory, and the I/O devices. Process and memory management is provided by the kernel server. I/O management is provided by the device server. Both of these are part of the kernel present on each machine, and are accessed via the standard message mechanism described above. They are special only in that they run in kernel mode and can get at the raw hardware.

Processes are organized into groups called **teams**. A team of processes share a common address space, and therefore must all run on the same processor. Application programs can make use of concurrency by running as a team of processes, each of which does part of the kernel. If one process in a team is blocked waiting for a reply to a message, the other ones are free to run. The kernel server is prepared to carry out operations such as creating new processes and teams, destroying processes and teams, reading and writing processes' states, and mapping processes onto memory.

All I/O in V is done using a uniform interface called the **V I/O protocol**. The protocol allows processes to read and write specific blocks on the device. This block orientation was chosen to provide idempotency. Terminal drivers must store the last block read and filter out duplicate requests in order to maintain the idempotency property. Implementation of byte streams is up to the users. The I/O protocol has proven general enough to handle disks, printers, terminals, and even a mouse.

4.3.4. Fault Tolerance

Since it was designed primarily for use in an interactive environment, V provides little in the way of fault tolerance. If something goes wrong, the user just does it again. However, V does address exception handling. Whenever a process causes an exceptional condition to occur, such as stack overflow or referencing nonexistent memory, the kernel detecting the error sends a specially formatted message to the **exception server**, which is outside the kernel. The exception server can then invoke a debugger to take over. This scheme does not require a process to make any advance preparation for being debugged, and in principle, can allow the process to continue execution

afterwards.

4.3.5. Services

Since most of the V workstations do not have a disk, the central file server plays a key role in the system. The file server is not part of the operating system. Instead, it is just an ordinary user program running on top of the V kernel. Internally it is structured as a team of processes. The main process handles directory operations, including opening files; subsidiary processes perform the actual read and write commands, so that when one of them blocks waiting for a disk block, the others can continue operation. The members of file server team share a common buffer cache, used to keep heavily used blocks in main memory.

The file system is a traditional hierarchical system, similar to that of Thoth [CHERITON 1982]. Each file has a file descriptor, similar to an i-node in UNIX, except that the file descriptors are gathered into an ordinary file which can grow as needed.

Extensive measurements have been made of the performance of the file server. As an indication, it takes 7.8 millisec to read a 1K block from the file server when the block is in the cache. This time includes the communication and network overhead. When the block must be fetched from the disk, the time is increased to 35.5 millisec. Given that the access time of the small Winchester disks used on personal computers is rarely better than 40 millsec, it is clear that the V implementation of diskless workstations with a fast (18 millisec) central file server is definitely competitive.

Other V servers include the print server, gateway server, and time server. Other servers are in the process of being developed.

4.3.6. Implementation

The V kernel has been up and running at Stanford since Sept. 1982. It runs on SUN Microsystems 68000-based workstations, connected by 3 Mbit/sec and 10 Mbit/sec Ethernets. The kernel is used as a base for a variety of projects at Stanford, including the research project on distributed operating systems. A great deal of attention has been paid to tuning the system to make it fast.

4.4. THE EDEN PROJECT

The goal of the Eden system [ALMES et al. 1985; BLACK 1985; BLACK 1983; JESSOP et al. 1982; LAZOWSKA et al. 1981], which is being developed at the University of Washington in Seattle under the direction of Guy Almes, Andrew Black, Ed Lazowska, and Jerre Noe, is to investigate logically integrated but physically distributed operating systems. The idea is to construct a system based on the principle of one user, one workstation (no processor pool), but with a high degree of systemwide integration. Eden is object oriented, with all objects accessed by capabilities, which are protected by the Eden kernel. Eden objects, in contrast to, say, Amoeba objects, contain not only passive data, but also one or more processes that carry out the operations

defined for the object. Objects are general: applications programmers can determine what operations their objects will provide. Objects are also mobile, but at any instant each object (and all the processes it contains) resides on a single workstation.

Much more than most research projects of this kind, Eden was designed top down. In fact, the underlying hardware and language was radically changed twice during the project, without causing too much redesign. This would have been much more difficult in a bottom-up, hardware-driven approach.

4.4.1. Communications Primitives

Communication in Eden uses "invocation," a form of remote procedure call. Programs are normally written in EPL, the Eden Programming Language, which is based on Concurrent Euclid. (The EPL translator is actually a preprocessor for Concurrent Euclid). To perform an operation on an object, say, *Lookup* on a directory object, the EPL programmer just calls *Lookup*, specifying a capability for the directory to be searched, the string to be searched for, and some other parameters.

The EPL compiler translates the call to *Lookup* to a call to a stub routine linked together with the calling procedure. This stub routine assembles the parameters and packs them in a standard form called ESCII (Eden Standard Code for Information Interchange), and then calls a lower level routine to transmit the function code and packed parameters to the destination machine.

When the message arrives at the destination machine, a stub routine there unpacks the ESCII message and makes a local call on *Lookup* using the normal EPL calling sequence. The reply proceeds analogously in the opposite direction. The stub routines on both sides are automatically generated by the EPL compiler.

The implementation of invocation is slightly complicated by the fact that an object may contain multiple processes. When one process blocks waiting for a reply, the other ones must not be affected. This problem is handled by splitting the invocation into two layers. The upper layer builds the message, including the capability for the object to be invoked and the ESCII parameters, passes it to the lower layer, and blocks the calling process until the reply arrives. The lower layer then makes a nonblocking call to the kernel to actually send the message. If other processes are active within the object they can now be run; if none are active, the object waits until a message arrives.

On the receiving side, a process within the invoked object will normally have previously executed a call announcing its willingness to perform some operation (e.g., *Lookup* in the above example) thereby blocking itself. When the *Lookup* message comes in, it is accepted by a special dispatcher process that checks to see which process, if any, is blocked waiting to perform the operation requested by the message. If a willing process is found, it runs and sends a reply, unblocking the caller. If no such process can be found, the message is queued until one becomes available.

4.4.2. Naming and Protection

Naming and protection in Eden is accomplished using the capability system. Data are encapsulated within objects, and are only accessible by invoking one of the operations defined by the object. To invoke an object, a process must have a valid capability. Thus there is a uniform naming and protection scheme throughout Eden.

Capabilities may be stored in any object. Directories provide a convenient mechanism for grouping capabilities together. Each directory entry contains the ASCII string by which the capability is accessed and the capability itself. Clients can only access the contents of a directory by invoking the directory object with one of the valid operations, which include: add entry, delete entry, lookup string, and rename capability. Capabilities are protected from forgery by the kernel, but users keep copies of capabilities for their own use; the kernel verifies them when they are used.

The basic protection scheme protects objects, using capabilities. Since all processes are embedded in objects, a process can be protected by restricting the distribution of capabilities to its object. The only way to obtain service from an object is by invoking the object with the proper capability, parameters, etc., all of which are checked by the kernel and EPL run-time system before the call is made.

4.4.3. Resource Management

Because no version of Eden runs on bare machines, most of the issues associated with low-level resource management have not yet been dealt with. Nevertheless, some resource management issues have been addressed. For example, when an object is created, the issue arises of where to put it. At present, it is just put on the same workstation as the object that created it unless an explicit request has been given to put it somewhere else.

Another issue that has received considerable attention is how to achieve concurrency within an object. From the beginning of the project it was considered desirable to allow multiple processes to be simultaneously active within an object. These processes all share a common address space, although each one has its own stack for local variables, procedure call/return information etc. Having multiple active processes within an object, coupled with the basic Eden semantics of remote invocations that block the caller but not the whole object, makes the implementation somewhat complicated. It is necessary to allow one process to block waiting for a reply without blocking the object as a whole. Monitors are used for synchronization. This multiprogramming of processes within an object is handled by a runtime system within that object, rather than by the kernel itself (as is done in Amoeba, and also in V). The experiences of Eden, Amoeba and V all seem to indicate that having cheap, "lightweight" processes that share a common address space is often useful [BLACK 1985].

Management of dynamic storage for objects has also been a subject of some work. Each object has a heap for its own internal use, for which the EPL compiler generates explicit allocate and deallocate commands. However, a different storage management scheme is used for objects themselves. When a

kernel creates an object, it allocates storage for the object from its own heap and gives the object its own address space. It also manages the user capabilities for the object in such a way that it is possible to systematically find all capabilities by scanning the kernel's data structures.

The system is periodically shut down and a garbage collector is started up to locate all objects for which no capability is outstanding. These objects are then discarded.

4.4.4. Fault Tolerance

The Eden kernel does not support atomic actions directly, although some services provide them to their clients. Invocations can fail with status CANNOT LOCATE OBJECT when the machine on which the invoked object resides crashes. On the other hand, Eden goes to a considerable length to make sure that objects are not totally destroyed by crashes. The technique used to accomplish this goal is to have objects checkpoint themselves periodically. Once an object has written a copy of its state to disk, a subsequent crash merely has the effect of resetting the object to the state it had at the most recent checkpoint. Checkpoints themselves are atomic, and this property can be used to build up more complex atomic actions.

By judicious timing of its checkpoints, an object can achieve a high degree of reliability. For example, within the user mail system, a mailbox object will checkpoint itself just after any letter is received or removed. Upon receipt of a letter, a mailbox can wait for confirmation of the checkpoint before sending an acknowledgement back to the sender, to ensure that letters are never lost due to crashes. One drawback of the whole checkpoint mechanism is that it is expensive: any change to an object's state, no matter how small, requires writing the entire object to the disk. The Eden designers acknowledge this as a problem.

Another feature of Eden that supports fault tolerance is the ability of the file system, when asked, to store an object as multiple copies on different machines (see below).

4.4.5. Services

The Eden file system maintains arbitrary objects. One particular object type, the BYTESTORE, implements linear files, as in UNIX. It is possible to set the "current position" anywhere in the file, and then read sequentially from that point. Unlike V and Amoeba, Eden does not have special machines dedicated as servers. Instead, each workstation can support file objects, either for the benefit of the local user or remote ones.

The model used for file service in Eden is quite different from the usual model of a file server, which manages some set of files and accepts requests from clients to perform operations on them. In Eden, each file (i.e., BYTESTORE object) contains within it the processes needed to handle operations on it. Thus, the file contains the server rather than the server containing the file as in most other systems.

Of course, actually having a process running for each file in existence would be unbearably expensive, so an optimization is used in the implementation. When a file is not open, its processes are dormant and consume no resources (other than the disk space for its checkpoint). Mailboxes, directories, and all other Eden objects work the same way. When an object is not busy with an invocation, the processes inside of it are put to sleep by checkpointing the whole object to the disk.

When a file is opened, a copy of the code for its internal processes is found, and the processes started up. Although all files on a given workstation share the same code, when the first file is opened on a workstation, the code may have to be fetched from another workstation.

The approach has advantages and disadvantages compared to the traditional one-file-server-for-all-files way of doing things. There are two main advantages. First, The complicated, multi-threaded file server code is eliminated: there *is* no file server. The processes within a BYTESTORE object are dedicated to a single file. Second, files can be migrated freely about all the nodes in the system, so that, for example, a file might be created locally, and then moved to a remote node where it will later be used.

The chief disadvantage is performance. All the processes needed for the open files consume resources, and fetching the code for the first file to be opened on a workstation is slow.

The Eden File System supports nested transactions [PU and NOE 1985]. When an atomic update on a set of files (or other objects) is to be carried out, the manager for that transaction first makes sure that all the new versions are safely stored on disk, then it checkpoints itself, and finally it updates the directory.

The transaction facility can be used to support replicated files [PU et al. 1986]. In the simplest case, a directory object maps an ASCII name onto the capability for that object. However, the system also has "repdirs," objects that map ASCII names onto sets of capabilities, for example, all the copies of a replicated file. Updating a replicated file is handled by a transaction manager, which uses a two-phase commit algorithm to update all the copies simultaneously. If one of the copies is not available for updating (e.g., its machine is down or the network is partitioned), a new copy of the file is generated, and the capability for the unreachable copy discarded. Sooner or later, the garbage collector will notice that the old copy is no longer in use and remove it.

We touched briefly on the mail server above. The mail system defines message, mailbox and address list objects, with operations to deliver mail, read mail, reply to mail, and so on.

The appointment calendar system is another example of an Eden application. It is used to schedule meetings, and runs in two phases. When someone proposes a meeting, a transaction is first done to mark the proposed time as "tentatively occupied" on all the participants' calendars. When a participant notices the proposed date, he or she can then approve or reject it. If all participants approve the meeting, it is "committed" by another transaction; if someone rejects the proposed appointment, the other participants are notified.

4.4.6. Implementation

Eden has had a somewhat tortuous implementation history. The initial version was designed to be written in Ada* on the Intel 432, a highly complex multiprocessor, fault-tolerant microprocessor chip ensemble. To make a long story short, neither the Ada compiler nor the 432 lived up the the project's expectations. To gather information for further design, a "throwaway" implementation was made on top of VMS on a VAX.

The VAX/VMS version, called Newark (because that was thought to be far from Eden), was written in Pascal and was not distributed (i.e., it ran on a single VA X). It supported multiple processes per object (VMS kernel processes), but did not have automatic stub generation. Furthermore, the whole implementation was rather cumbersome, so it was then decided to design a programming language which would provide automatic stub generation, better type checking, and a more convenient way of dealing with concurrency.

This re-evaluation led to EPL and a new implementation on top of UNIX instead of VMS. Subsequently, Eden was ported to 68000-based workstations (SUNs), also on top of UNIX, rather than on the bare hardware (and in contrast to the Cambridge system, V, and Amoeba, all of which run on bare 68000s). The decision to put UNIX on the bottom, instead of the top (as was done with Amoeba) made system development easier and assisted users in migrating from UNIX to Eden. The price that has been paid is poor performance, and a fair amount of effort spent trying to convince UNIX to do things against its will.

4.5. Comparison of the Cambridge, Amoeba, V, and Eden systems

Our four example systems have many aspects in common, but also differ in some significant ways. In this section we will summarize and compare the four systems with respect to the main design issues we have been looking at.

4.5.1. Communication Primitives

All four systems use an RPC-like mechanism (as opposed to an ISO OSI communication-oriented mechanism).

The Cambridge mechanism is the simplest, using the single shot protocol with a 2K request packet and a 2K reply packet for most client-server communication. A byte stream protocol is also available.

Amoeba uses a similar REQUEST-REPLY mechanism, but allows messages up to 32K bytes (with the kernel handling message fragmentation and reassembly), as well as acknowledgements and timeouts, thus providing user programs with a more reliable and simpler interface.

V also uses a REQUEST-REPLY mechanism, but messages longer than an Ethernet packet are dealt with by having the sender include a sort of "capability" for a message segment in the REQUEST packet. Using this "capability," the receiver can fetch the rest of the message, as needed. For efficiency, the

* Ada is a Trademark of the U.S. Dept. of Defense

first 1K is piggybacked onto the REQUEST itself.

Eden comes closest to a true RPC mechanism, including having a language and compi ler with automatic stub generation and a minilanguage for parameter passing. None of the four examples attempts to guarantee that remote calls will be executed exactly once.

4.5.2. Naming and Protection

All four systems use different schemes for naming and protection. In the Cambridge system, a single name server process maps symbolic service names onto (node, process identifier) pairs so the client will know where to send the request. Protection is done by the active name table, which keeps track of the authorization status of each logged in user.

Amoeba has a single mechanism for all naming and protection—sparse capabilities. Each capability contains bits specifying which operations on the object are allowed and which are not. The rights are protected cryptographically, so user programs can manipulate them directly; they are not stored in the kernel. ASCII string to capability mapping and capability storage are handled by directory servers for convenience.

Eden also uses capabilities, but these are not protected by sparseness or encryption, so they must be protected by the kernel. A consequence of this decision is that all the kernels must be trustworthy. The Amoeba cryptographic protection scheme is less restrictive on this point.

V has naming at three levels: processes have pids, kernels have ASCII to pid mappings, and servers use a context mechanism to relate symbolic names to a given context.

4.5.3. Resource Management

Resource management is also handled quite differently on all four systems. In the Cambridge system, the main resource is the processor bank. A resource manager is provided to allocate machines to users. Generally, this allocation is fairly static—upon log in a user is allocated one machine for the duration of the login session, and this is the only machine the user uses during the session. He may load any operating system he chooses in this machine.

Amoeba also has a pool of processors, but these are allocated dynamically. A user running "make" might be allocated 10 processors to compile 10 files; afterwards, all the processors would go back into the pool. Amoeba also provides a way for processes to create segments on any machine (assuming the proper capability can be shown) and for these segments to be forged into processes. Amoeba is unique among the four systems in that it has a bank server that can allow servers to charge for services and to limit resource usage by accounting for it.

In V, each processor is dedicated as either a workstation or a server, so processors are not resources to be dynamically allocated. Each V kernel manages its own local resources; there is no system-wide resource management.

Eden has been built on top of existing operating systems, so most of the issues of resource management are done by the underlying operating system.

The main issue remaining for Eden is allocating and deallocating storage for objects.

4.5.4. Fault Tolerance

None of the four systems go to great lengths to make themselves fault tolerant, for example, none support atomic actions as a basic primitive. All four (with the possible exception of Eden) were designed with the intention of actually being used, so that the inherent tradeoff between performance and fault tolerance tended to get resolved in favor of performance.

In the Cambridge system, the only concession to fault tolerance is a feature in the ring interface to allow a machine to be remotely reset by sending a special packet to the interface. There is also a small server that helps get the servers started up.

Amoeba provides some fault tolerance through its boot server, with which processes can register. The boot server pools the registered processes periodically, and finding one that fails to respond, requests a new processor and downloads the failed program to it. This strategy does not retrieve the processes that were killed when a machine went down, but it does automatically ensure than no key service is ever down for more than, say, 30 seconds.

V does not address the problem of fault tolerance at all.

Of the four systems, Eden makes the most effort to provide a higher degree of reliability than provided by the bare hardware. The main tool used is checkpointing complete objects from time to time. If a processor crashes, each of its objects can be restored to the state it had at the time of the last checkpoint. Unfortunately, only entire objects can be checkpointed, making checkpointing a slow operation, thus discouraging its frequent use.

4.5.5. Services

The file systems used by Cambridge, Amoeba, V, and Eden are all quite different. The Cambridge system has two servers, the universal file server, and the filing machine, which was added later to improve the performance by providing a large buffer cache. The universal file server supports a primitive flat file, with no directory structure, this being provided by the filing machine or the user machines. The universal file server has regular and special files, the latter of which can be updated atomically.

Amoeba has several file systems. One of them is compatible with UNIX, to allow UNIX applications to run on Amoeba. Another one, FUSS, supports multiversion, multiserver, tree structured, immutable files with atomic commit. Directory servers map ASCII names to capabilities, thus allowing an arbitrary graph of files and directories to be constructed.

V has a traditional file server similar to UNIX. It is based on the earlier Thoth system.

Eden has no file server at all in the usual sense. Instead, each file object has embedded in it a process that acts like a private file server for that one file. Like Amoeba, Eden has separate directory servers that map ASCII strings onto capabilities, and provides the ability to map one string onto several files,

thus providing for file replication. All four systems have a heterogeneous variety of other services (e.g., print, mail, bank).

5. Summary

Distributed operating systems are still in an early phase of development, with many unanswered questions, and relatively little agreement among workers in the field about how things should be done. Many experimental systems use the client-server model with some form of remote procedure call as the communication base, but there are also systems built on the connection model. Relatively little has been done on distributed naming, protection, and resource management, other than building straightforward name servers and process servers. Fault tolerance is an up and coming area, with work progressing in redundancy techniques and atomic actions. Finally, a considerable amount of work has gone into the construction of file servers, print servers, and various other servers, but here too there is much work to be done. The only conclusion we draw is that distributed operating systems will be an interesting and fruitful area of research for a number of years to come.

Acknowledgements

We would like to thank Andrew Black, Dick Grune, Sape Mullender, and Jennifer Steiner for their critical reading of the manuscript.

References

Adams, C.J., Adams, G.C., Waters, A.G., Leslie, I., Kirk, P. "Protocol Architecture of the UNIVERSE Project," *Proc. Sixth Int'l Conf. on Computer Communication,* , London, pp. 379-383, 1982.

Almes, G.T., Black, A.P., Lazowska, E.D. and Noe, J.D. "The Eden System: A Technical Rev.," *IEEE Trans. Softw. Engineering*, vol. SE-11, pp. 43-59, Jan. 1985.

Anderson, T., and Lee., P.A.: *Fault Tolerance, Principles and Practice*, London: Prentice-Hall, Int'l, 1981.

Avizienis, A. and Chen, L. "On the Implementation of N-version Programming for Software Fault-Tolerance During Execution," *Proc. COMPSAC*, IEEE, pp. 149-155, 1977.

Avizienis, A. and Kelly, J. "Fault Tolerance by Design Diversity," *Computer*, vol. 17, pp. 66-80, Aug. 1984.

Bal, H.E., van Renesse, R. and Tanenbaum, A.S. "A Distributed, Parallel, Fault Tolerant Computing System," Report IR-106, Dept. of Math. and Comp. Sci., Vrije Univ., Oct. 1985.

Ball, J.E., Feldman, J., Low, R., Rashid, R. and Rovner, P. "RIG, Rochester's Intelligent Gateway: System Overview," *IEEE Trans. Softw. Engineering*, vol. SE-2, pp. 321-329, Dec. 1976.

Barak, A. and Shiloh, A. "A Distributed Load-balancing Policy for a Multicomputer," *Software—Practice & Experience*, vol. 15, pp. 901-913, Sept. 1985.

BIRMAN, K.P. and ROWE, L.A. "A Local Network Based on the UNIX Operating System," *IEEE Trans. Softw. Eng.*, vol. SE-8, pp. 137-146, March 1982.

BIRRELL, A.D. "Secure Communication Using Remote Procedure Calls," *ACM Trans. Comput. Syst.*, vol. 3, pp. 1-14, Feb. 1985.

BIRRELL, A.D. and NELSON, B.J. "Implementing Remote Procedure Calls," *ACM Trans. Comput. Systems*, vol. 2, pp. 39-59, Feb. 1984.

BIRRELL, A.D., LEVIN, R., NEEDHAM, R.M. and SCHROEDER, M. "Experience with Grapevine: The Growth of a Distributed System," *ACM Trans. Comput. Syst.*, vol. 2, pp. 3-23, Feb. 1984.

BIRRELL, A.D., LEVIN, R., NEEDHAM, R.M. and SCHROEDER, M. "Grapevine: An Exercise in Distributed Computing," *Commun. ACM*, vol. 25, pp. 260-274, April 1982.

BIRRELL, A.D. and NEEDHAM, R.M. "A Universal File Server," *IEEE Trans. Softw. Eng.*, vol. SE-6, pp. 450-453, Sept. 1980.

BLACK, A.P. "Supporting Distributed Applications: Experience with Eden," *Tenth Symp. Oper. Syst. Prin.*, ACM, pp. 181-193, Dec. 1985.

BLACK, A.P. "An Asymmetric Stream Communications System," *Proc. Ninth Symp. Operating Syst. Prin.*, ACM, pp. 4-10, Oct. 1983.

BOGGS, D.R., SCHOCH, J.F., TAFT, E.A. and METCALFE, R.M. "Pup: An Internetwork Architecture," *IEEE Trans. Commun.*, vol. C-28, pp. 612-624, April 1980.

BORG, A., BAUMBACH, J. and GLAZER, S. "A Message System Supporting Fault Tolerance," *Proc. Ninth Symp. Operating Syst. Prin.*, ACM, pp. 90-99, 1983.

BROWN, M.R., KOLLING, K.N. and TAFT, E.A. "The Alpine File System," *ACM Trans. Comput. Syst.*, vol. 3, pp. 261-293, Nov. 1985.

BROWNBRIDGE, D.R., MARSHALL, L.F. and RANDELL, B. "The Newcastle Connection- or UNIXES or the World Unite!," *Software—Practice & Experience*, vol. 12, pp. 1147-1162, Dec. 1982.

BRYANT, R.M. and FINKEL, R.A. "A Stable Distributed Scheduling Algorithm" *Proc. 2nd Int'l Conf. on Distributed Comput. Syst.*, IEEE, pp. 314-323, April 1981.

CHANDY, K.M., MISRA, J. and HAAS, L.M. "Distributed Deadlock Detection," *ACM Trans. Comput. Syst.*, vol. 1, pp. 145-156, May 1983.

CHERITON, D.R. "An Experiment Using Registers for Fast Message-Based Interprocess Communication," *Operating Systems Rev.*, vol 18, pp. 12-20, Oct. 1984.

CHERITON, D.R "The V Kernel: A Software Base for Distributed Systems," *IEEE Software*, vol. 1, pp. 19-42, April 1984.

CHERITON, D.R. *The Thoth System: Multi-process Structuring and Portability*, New York: American Elsevier, 1982.

CHERITON, D.R., MALCOLM, M.A., MELEN, L.S. and SAGER, G.R. "Thoth, A Portable Real-Time Operating System," *Commun. ACM*, vol. 22, pp. 105-115, Feb. 1979.

CHERITON, D.R. and MANN, T.P. "Uniform Access to Distributed Name

Interpretation in the V System," *Proc. Fourth Int'l Conf. on Distributed Comput. Syst.*, IEEE, pp. 290-297, 1984.

CHERITON, D.R. and ZWAENEPOEL, W. "One-to-Many Interprocess Communication in the V-System," *ACM Sigcomm 84 Symp.*, 1984a.

CHERITON, D.R. and ZWAENEPOEL, W. "The Distributed V Kernel and its Performance for Diskless Workstations," *Proc. Ninth Symp. Operating Syst. Prin.*, ACM, pp. 128-140, 1984.

CHESSON, G. "The Network UNIX System," *Proc. Fifth Symp. Operating Syst. Prin.*, ACM, pp. 60-66, Nov. 1975.

CHU, W.W., HOLLOWAY, L.J., MIN-TSUNG, L., EFE, K. "Task Allocation in Distributed Data Processing," *Computer*, vol. 13, pp. 57-69, Nov. 1980.

CHOW, T.C.K. and ABRAHAM, J.A. "Load Balancing in Distributed Systems," *IEEE Trans. Softw. Engineering*, vol. SE-8, pp. 401-412, July 1982.

CHOW, Y.C. and KOHLER, W.H. "Models for Dynamic Load Balancing in Heterogeneous Multiple Processor Systems," *IEEE Trans. Comput.*, vol. C-28, pp. 354-361, May 1979.

CURTIS, R.S. and WITTIE, L.D. "Global Naming in Distributed Systems," *IEEE Software*, vol. 1, pp. 76-80, 1984.

DALAL, Y.K. "Broadcast Protocols in Packet Switched Computer Networks," Ph. D. Thesis, Stanford Univ., 1977.

DELLAR, C. "A File Servers for a Network of Low-Cost Personal Microcomputers," *Software-Practice & Experience*, vol. 12, pp. 1051-1068, Nov. 1982.

DENNIS, J.B. and VAN HORN, E.C. "Programming Semantics for Multiprogrammed Computations," *Commun. ACM*, vol. 9, pp. 143-154, March 1966.

DEWITT, D.J., FINKEL, R.A. and SOLOMON, M. "The CRYSTAL Multicomputer: Design and Implementation Experience," TR-553, University of Wisconsin, Sep. 1984.

DION, J. "The Cambridge File Server," *Operating Syst. Rev.*, vol. 14, pp. 41-49, Oct. 1980.

EFE, K., "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *Computer*, vol. 15, pp. 50-56, June 1982.

ESWARAN, K.P., GRAY, J.N., LORIE, J.N. and TRAIGER, I.L. "The Notions of Consistency and Predicate Locks in a Database System," *Commun. ACM*, vol. 19, pp. 624-633, Nov. 1976.

FARBER, D.J. and LARSON, K.C. "The System Architecture of the Distributed Computer System-The Communications System," *Symp. Computer Netw.*, Polytechnic Institute of Brooklyn, April 1972.

FINKEL, R.A., SOLOMON, M.H. and TISCHLER, R. "The Roscoe Resource Manager," *COMPCON 79 Digest of Papers*, IEEE, pp. 88-91, Feb. 1979.

FITZGERALD, R. and RASHID, R. "The Integration of Virtual Memory Management and Interprocess Communication in Accent," *Proc. 10th Symp. Operating Syst. Prin.*, ACM, pp. 13-14, Dec. 1985.

FRIDRICH, M. and OLDER, W. "HELIX: The Architecture of a Distributed File System," *Proc. Fourth Int'l. Conf. on Distributed Comput. Syst.*, IEEE, pp. 422-431, 1984.

FRIDRICH, M. and OLDER, W. "The Felix File Server," *Proc. Eighth Symp. Operating Syst. Prin.*, ACM, pp. 37-44, 1981.

GAGLIANELLO, R.D. and KATSEFF, H.P. "Meglos: An Operating System for a Multiprocessor Environment," *Proc. Fifth Int'l. Conf. on Distributed Comput. Syst.*, IEEE, pp. 35-42, May 1985.

GLIGOR, V.D. and SHATTUCK, S.H. "Deadlock Detection in Distributed Systems," *IEEE Trans. Softw. Eng.*, vol. SE-6, pp. 435-440, Sept. 1980.

GYLYS, V.B. and EDWARDS, J.A. "Optimal Partitioning of Workload for Distributed Systems," *COMPCON,* pp. 353-357, Sept. 1976.

HWANG, K., CROFT, W.J., GOBLE, G.H., WAH, B.W., BRIGGS, F.A., SIMMONS, W.R. and COATES, C.L. "A UNIX-Based Local Computer Network," *Computer*, vol. 15, pp. 55-66, April 1981.

ISLOOR, S.S. and MARSLAND, T.A. "An Effective On-line Deadlock Detection Technique for Distributed Database Management Systems," *Proc. COMPSAC*, IEEE, pp. 283-288, 1978.

JEFFERSON, D.R. "Virtual Time," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 404-425, July 1985.

JENSEN, E. D. "The Honeywell Experimental Distributed Processor-An Overview of its Objective, Philosophy and Architectural Facilities," *Computer*, vol. 11, pp. 28-38, Jan. 1978.

JESSOP, W.H., JACOBSON, D.M., NOE, J.D., BAER, J.-L. and PU, C. "The Eden Transaction-Based File System," *Proc. 2nd Symp. Reliability in Distr. Software and Database Syst.*, pp. 163-169, July 1982.

KRUEGER, P. and FINKEL, R.A. "An Adaptive Load Balancing Algorithm for a Multicomputer," Computer Science Dept., Univ. of Wisconsin, 1983.

LAMPORT, L. "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems," *ACM Trans. Program. Lang. Syst.*, vol. 6, pp. 254-280, April 1984.

LAMPORT, L. "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, pp. 558-565, July 1978.

LAMPSON, B.W. "Atomic Transactions," in *Distributed Systems - Architecture and Implementation*, Berlin: Springer-Verlag, pp. 246-265, 1981

LAZOWSKA, E.D., LEVY, H.M., ALMES, G.T., FISCHER, M.J., FOWLER, R.J. and VESTAL, S.C. "The Architecture of the Eden System," *Proc. Eighth Symp. Operating Syst. Prin.*, pp. 148-159, Dec. 1981

LEVY, H.M. *Capability-Based Computer Systems*, Maynard, Mass.: Digital Press, 1984.

LISKOV, B. "Overview of the Argus Language and System," Programming Methodology Group Memo 40, MIT Lab. for Comp. Sci., Feb 1984.

LISKOV, B. "On Linguistic Support for Distributed Programs," *IEEE Trans. Softw. Eng.*, vol. SE-8, pp. 203-210, May 1982.

LISKOV, B. and SCHEIFLER, R. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Trans. Prog. Lang. Syst.*, vol. 5, pp. 381-404, July 1983. ACM, pp. 7-19, Jan. 1982.

LO, V.M. "Heuristic Algorithms for Task Assignment in Distributed Systems," *Proc. Fourth Int'l Conf. on Distributed Comput. Syst.*, IEEE, pp. 30-39, 1984.

LUDERER, G.W.R., CHE, H., HAGGERTY, J.P., KIRSLIS, P.A. and MARSHALL, W.T. "A Distributed UNIX System Based on a Virtual Circuit Switch," *Proc. Eighth Symp. Operating Syst. Prin.*, ACM, pp. 160-168, 1981.

MAMRAK, S.A., MAURATH, P., GOMEZ, J., JANARDAN, S. and NICHOLAS, C. "Guest Layering Distributed Processing Support on Local Operating Systems," *Proc. 3rd Int'l Conf. on Distributed Comput. Syst.*, IEEE, pp. 854-859.

MARZULLO, K. and OWICKI, S. "Maintaining the Time in a Distributed System," *Operating Syst. Rev.*, vol. 19 pp. 44-54, July 1985.

MENASCE, D. and MUNTZ, R. "Locking and Deadlock Detection in Distributed Databases," *IEEE Trans. Softw. Eng.*, vol. SE-5, pp. 195-202, May 1979.

MILLSTEIN, R.E. "The National Software Works," *Proc. ACM Ann. Conf.*, pp. 44-52, 1977.

MITCHELL, J.G. and DION, J. "A Comparison of Two Network-Based File Servers," *Commun. ACM*, vol. 25, pp. 233-245, April 1982.

MOHAN, C.K. and WITTIE, L.D. "Local Reconfiguration of Management Trees in Large Networks," *Proc. Fifth Int'l Conf. on Distributed Comput. Syst.*, IEEE, pp. 386-393, May 1985.

MULLENDER, S.J. and TANENBAUM, A.S. "The Design of a Capability-Based Distributed Operating System," *Computer Journal*, (to appear in 1986).

MULLENDER, S.J. and TANENBAUM, A.S. "A Distributed File Service Based on Optimistic Concurrency Control," *Proc. Tenth Symp. Operating Syst. Prin.*, ACM, pp. 51-62, 1985.

MULLENDER, S.J. and TANENBAUM, A.S. "Protection and Resource Control in Distributed Operating Systems," *Computer Networks*, vol 8, pp. 421-432, Nov. 1984.

NEEDHAM, R.M. and HERBERT, A.J. *The Cambridge Distributed Computing System*, Reading, Mass: Addison-Wesley, 1982.

NELSON, B.J. "Remote Procedure Call," Tech. Rep. CSL-81-9, Xerox PARC, 1981.

OBERMARCK, R. "Distributed Deadlock Detection Algorithm," *ACM Trans. Database Syst.*, vol. 7, pp. 187-208, June 1982.

OKI, B.M., LISKOV, B.H. and SCHEIFLER, R.W. "Reliable Object Storage to Support Atomic Actions," *Proc. 10th Symp. Operating Sys. Prin.*, pp. 147-159, Dec. 1985.

OUSTERHOUT, J.K. "Scheduling Techniques for Concurrent Systems," *Proc. 3rd Int'l Conf. on Distributed Comput. Syst.*, IEEE, pp. 22-30, 1982.

PASHTAN, A. "Object Oriented Operating Systems: An Emerging Design Methodology," *Proc. ACM National Conf.*, pp. 126-131, 1982.

POPEK, G., WALKER, B., CHOW, J., EDWARDS, D., KLINE, C., RUDISIN and G., THIEL, G. "LOCUS A Network Transparent, High Reliability Distributed System," *Proc. Eighth Symp. Operating Syst. Prin.*, ACM, pp. 160-168, 1981

POWELL, M.L. and MILLER, B.P. "Process Migration in DEMOS/MP," *Proc. Ninth Symp. Operating Syst. Prin.*, ACM, pp. 110-119, 1983.

POWELL, M.L. and PRESOTTO, D.L. "Publishing-A Reliable Broadcast Communication Mechanism," *Proc. Ninth Symp. Operating Syst. Prin.*, ACM, pp. 100-109, 1983.

PU, C. and NOE, J.D. "Nested Transactions for General Objects," Report TR-85-12-03. Univ. of Washington, Seattle, WA, 1985.

PU, C., NOE, J.D. and PROUDFOOT, A. "Regeneration of Replicated Objects: A Technique and its Eden Implementation," *Proc. Second Int'l Conf. on Data Engineering*, pp. 175-187, Feb. 1986.

RASHID, R.F. and ROBERTSON, G.G. "Accent: A Communication Oriented Network Operating System Kernel," *Proc. Eighth Symp. Operating Syst. Prin.*, ACM, pp. 64-75, 1981.

REED, D.P. "Implementing Atomic Actions on Decentralized Data," *ACM Trans. Comput. Syst.*, vol. 1, pp. 3-23, Feb. 1983

REED, D.P. and SVOBODOVA, L. "SWALLOW: A Distributed Data Storage System for a Local Network," In *Local Networks for Computer Communications*, A. West and P. Janson (eds.) North-Holland Publ., Amsterdam, pp. 355-373, 1981.

REIF, J.H. and SPIRAKIS, P.G. "Real-Time Synchronization of Interprocess Communications," *ACM Trans. Program. Lang. Syst.*, vol. 6, pp. 215-238, April 1984.

RITCHIE, D.M. and THOMPSON, K. "The UNIX Time-sharing System," *Commun. ACM*, pp. 365-375, July 1974.

SALTZER, J.H., REED, D.P. and CLARK, D.D. "End-to-End Arguments in System Design," *ACM Trans. Comput. Syst.*, vol. 2, pp. 277-278, Nov. 1984.

SATYANARAYANAN, M., HOWARD, J., NICHOLS, D., SIDEBOTHAM, R., SPECTOR, A. and WEST, M. "The ITC Distributed File System: Principles and Design," *Proc. Tenth Symp. Operating Syst. Prin.*, ACM, pp. 35-50, 1985.

SCHROEDER, M., GIFFORD, D. and NEEDHAM, R. "A Caching File System for a Programmer's Workstation," *Proc. Tenth Symp. Operating Syst. Prin.*, ACM, pp. 25-34, 1985.

SOLOMON, M.H. and FINKEL, R.A. "ROSCOE: A Multimicrocomputer Operating System," *Proc. 2nd Rocky Mtn. Symp. Microcomputers*, pp. 201-210, Aug. 1978.

SOLOMON, M.H. and FINKEL, R.A. "The Roscoe Distributed Operating System," *Proc. Seventh Symp. Operating Syst. Prin.*, ACM, pp. 108-114, 1979.p

SMITH, R. "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver," *Proc. 1st Int'l Conf. Distributed Comput. Syst.*, IEEE, pp. 185-192, 1979.

SPECTOR, A.Z. "Performing Remote Operations Efficiently on a Local Computer Network," *Commun. ACM*, vol. 25, pp. 246-260, April 1982.

STANKOVIC, J.A. and SIDHU, I.S. "An Adaptive Bidding Algorithm for Processes, Clusters, and Distributed ups," *Proc. Fourth Int'l Conf. on Distributed Comput. Syst.*, IEEE, pp. 49-59, 1984.

STONEBRAKER, M. "Operating System Support for Database Management," *Commun. ACM*, vol. 24, pp. 412-418, July 1981.

STONE, H.S. "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. Softw. Engineering*, vol. SE-3, pp. 88-93, Jan. 1977.

STONE, H.S. "Critical Load Factors in Distributed Computer Systems," *IEEE Trans. Softw. Eng.*, vol. SE-4, pp. 254-258, May 1978.

STONE, H.S. and BOKHARI, S.H. "Control of Distributed Processes," *Computer*, vol. 11, pp. 97-106, July 1978.

STURGIS, H.E., MITCHELL, J.G. and ISRAEL, J. "Issues in the Design and Use of a Distributed File System," *Operating Systems Rev.*, vol. 14, pp. 55-69, July 1980.

SVENTEK, J., GREIMAN, W., O'DELL, M. and JANSEN, A. "Token Ring Local Networks-A Comparison of Experimental and Theoretical Performance," Lawrence Berkeley Lab. Report 16254, 1983.

SVOBODOVA, L. "File Servers for Network-Based Distributed Systems," *Computing Surveys*, vol. 16, pp. 353-398, Dec. 1984.

SVOBODOVA, L. "A Reliable Object-Oriented Data Repository for a Distributed Computer System," *Proc. Eighth Symp. Operating Syst. Prin.*, ACM, pp. 47-58, 1981.

SWINEHART, D., MCDANIEL, G. and BOGGS, D. "WFS: A Simple Shared File System for a Distributed Environment," *Proc. Seventh Symp. Operating Syst. Prin.*, ACM, pp. 9-17, 1979.

TANENBAUM, A.S. and MULLENDER, S.J. "Operating System Requirements for Distributed Data Base Systems," in *Distributed Data Bases*, Schneider, H.-J. (ed.), North-Holland, pp. 105-114, 1982.

TANENBAUM, A.S., MULLENDER, S.J. and VAN RENESSE, R. "Using Sparse Capabilities in a Distributed Operating System," *Proc. Sixth Int'l Conf. on Distributed Computer Systems*, IEEE, 1986.

VAN TILBORG, A.M. and WITTIE, L.D. "Wave Scheduling: Distributed Allocation of Task Forces in Network Computers," *Proc. 2nd Int'l Conf. on Distributed Comput. Syst.*, IEEE, pp. 337-347, 1981.

WALKER, B., POPEK, G., ENGLISH, R., KLINE, C. and THIEL, G. "The LOCUS Distributed Operating System," *Proc. Ninth Symp. Operating Syst. Prin.*, ACM, pp. 49-70, 1983.

WAMBECQ, A. "NETIX: A Network-Using Operating System, Based on UNIX Software," Proc. NFWO-ENRS Contact Group, Leuven, Belgium, March 1983.

WEINSTEIN, M.J., PAGE, T.W., JR., LIVESEY, B.K. and POPEK, G.J. "Transactions and Synchronization in a Distributed Operating System," *Proc. 10th Symp. Oper. Syst. Prin.*, pp. 115-125, Dec. 1985.

WILKES, M.V. and NEEDHAM, R.M. "The Cambridge Model Distributed System," *Operating Systems Rev.*, vol. 14, pp. 21-29, Jan. 1980.

WITTIE, L. and CURTIS, R. "Time Management for Debugging Distributed Systems" *Proc. Fifth Int'l Conf. on Distributed Comput. Syst*, IEEE, pp. 549-551, May 1985.

WITTIE, L.D. and VAN TILBORG, A.M. "MICROS, A Distributed Operating System for MICRONET, A Reconfigurable Network Computer," *IEEE Trans. Comput.*, vol C-29, pp. 1133-1144, Dec. 1980.

WUPIT, A. "Comparison of UNIX Network Systems," *ACM Conf. on Personal and Small Computers,* ACM, pp. 99-108, 1983.

ZIMMERMANN, H. "OSI Reference Model-The ISO Model of Architecture for Open Systems Interconnection," *IEEE Trans. Commun.*, vol. COM-28, pp. 425-432, April 1980.

ZIMMERMAN, H., BANINO, J.-S., CARISTAN, A., GUILLEMONT, M. and MORISSET, G. "Basic Concepts for the Support of Distributed Systems: The Chorus Approach," *Proc. 2nd Int'l Conf. on Distributed Comput. Syst.*, IEEE, pp. 60-66, 1981.

The Design of a Capability-Based Distributed Operating System

Sape J. Mullender
Centre for Mathematics and Computer Science
Amsterdam, The Netherlands

Andrew S. Tanenbaum
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Fifth generation computer systems will use large numbers of processors to achieve high performance. In this paper a capability-based operating system designed for this environment is discussed. Capability-based operating systems have traditionally required large, complex kernels to manage the use of capabilities. In our proposal, capability management is done entirely by user programs without giving up any of the protection aspects normally associated with capabilities. The basic idea is to use one-way functions and encryption to protect sensitive information. Various aspects of the proposed system are discussed.

1980 Mathematics Subject Classification: 68A05, 68B20.
1982 CR Categories: C.2.2, C.2.4, D.4.4, D.4.6.
Keywords & Phrases: distributed operating systems, capabilities, connectionless protocols, transaction-oriented protocols, protection, accounting, file systems, service model.

1. Introduction

Fifth generation computers must be fast, reliable, and flexible. One way to achieve these goals is to build them out of a small number of basic modules that can be assembled together to realize machines of various sizes. The use of multiple modules can make the machines not only fast, but also achieve a substantial amount of fault tolerance. The system architecture and software for such machines are described below.

The Design of a Capability-Based Distributed Operating System
S. J. Mullender and A. S. Tanenbaum
The Computer Journal
Vol. 29, No. 4, pp. 289-300
March 1986

1.1. System architecture

The price of processors and memory is decreasing at an incredible rate. Extrapolating from the current trend, it is likely that a single board containing a powerful CPU, a substantial fraction of a megabyte of memory, and a fast network interface will be available for a manufacturing cost of less than 100 in 1990. Our intention is therefore to do research on the architecture and software of machines built up of a large number of such modules.

In particular, we envision three classes of machines: (1) personal computers consisting of a high-quality bit-map display and a few processor-memory modules; (2) departmental machines consisting of hundreds of such modules; and (3) large mainframes consisting of thousands of them. The primary difference between these machines is the number of modules, rather than the type of the modules. In principle, any of these machines can be gracefully increased in size to improve performance by adding new modules or decreased in size to allow removal and repair of defective modules. The software running on the various machines should be in essence identical. Furthermore, it should be possible to connect different machines together to form even larger machines and to partition existing machines into disjoint pieces when necessary, all in a way transparent to the user level software.

This model is superior to the oft-proposed "Personal Computer Model" (as exemplified by XEROX PARC), in a number of ways. In the personal computer model, each user has a dedicated minicomputer, complete with disks, in his office or at home. Unfortunately, when people work together on large projects, having numerous local file systems can lead to multiple, inconsistent copies of many programs. Also, the noise generated by disks in every office, and the maintenance problems generated by having machines spread all over many buildings can be annoying.

Furthermore, computer usage is very bursty: most of the time the user does not need any computing power, but once in a while he may need a very large amount of computing power for a short time (e.g., when recompiling a program consisting of 100 files after changing a basic shared declaration). The fifth generation computer we propose is especially well suited to bursty computation. When a user has a heavy computation to do, an appropriate number of processor-memory modules are temporarily assigned to him. When the computation is completed, they are returned to the idle pool for use by other users. This contrasts with the Cambridge Distributed Operating System [Needham82], which also has a "processor bank," but assigns a processor to a user for the duration of a login session.

1.2. System software

A machine of the type described above requires radically different system software than existing machines. Not only must the operating system effectively use and manage a very large number of processors, but the communication and protection aspects are very different from those of existing systems.

Traditional networks and distributed systems are based on the concept of

two processes or processors communicating via connections. The connections are typically managed by a hierarchy of complex protocols, usually leading to complex software and extreme inefficiency. (An effective transfer rate of 0.1 megabit/sec over a 10 megabit/sec local network, which is only 1% utilization, is frequently barely achievable.)

We reject this traditional approach of viewing a distributed system as a collection of discrete processes communicating via multilayer (e.g., ISO) protocols, not only because it is inefficient, but because it puts too much emphasis on specific processes, and by inference, on processors. Instead we propose to base the software design on a different conceptual modelthe object model. In this model, the system deals with abstract objects, each of which has some set of abstract operations that can be performed on it.

Associated with each object are one or more "capabilities" [Dennis66] which are used to control access to the object, both in terms of who may use the object and what operations he may perform on it. At the user level, the basic system primitive is performing an operation on an object, rather than such things as establishing connections, sending and receiving messages, and closing connections. For example, a typical object is the file, with operations to read and write portions of it.

The object model is well-known in the programming languages community under the name of "abstract data type" [Liskov74]. This model is especially well-suited to a distributed system because in many cases an abstract data type can be implemented on one of the processor-memory modules described above. When a user process executes one of the visible functions in an abstract data type, the system arranges for the necessary underlying message transport from the user's machine to that of the abstract data type and back. The header of the message can specify which operation is to be performed on which object. This arrangement gives a very clear separation between users and objects, and makes it impossible for a user to directly inspect the representation of an abstract data type by bypassing the functional interface.

A major advantage of the object or abstract data type model is that the semantics are inherently location independent. The concept of performing an operation on an object does not require the user to be aware of where objects are located or how the communication is actually implemented. This property gives the system the possibility of moving objects around to position them close to where they are frequently used. Furthermore, the issue of how many processes are involved in carrying out an operation, and where they are located is also hidden from the user.

It is frequently convenient to *implement* the object model in terms of clients (users) who send messages to services [Cheriton83, Needham82, Ball79]. A service is defined by a set of commands and responses. Each service is handled by one or more server processes that accept messages from clients, carry out the required work, and send back replies. The design of these servers and the design of the protocols they use form an important part of the system software of our proposed fifth generation computers.

As an example of the problems that must be solved, consider a file server.

Among other design issues that must be dealt with are how and where information is stored, how and when it is moved, how it is backed up, how concurrent reads and writes are controlled, how local caches are maintained, how information is named, and how accounting and protection are accomplished. Furthermore, the internal structure of the service must be designed: how many server processes are there, where are they located, how and when do they communicate, what happens when one of them fails, how is a server process organized internally for both reliability and high performance, and so on. Analogous questions arise for all the other servers that comprise the basic system software.

2. Communication primitives and protocols

In the literature about computer networks, one finds much discussion of the ISO OSI reference model [Zimmermann80] these days. It is our belief that the price that must be paid in terms of complexity and performance in order to achieve an "open" system in the ISO sense is much too high, so we have developed a much simpler set of communication primitives, which we will now describe.

2.1. Transaction vs. stream communication

Most distributed systems have a connection mechanism that is based on the idea of two processes going to some effort to set up a connection, using the connection, and then tearing it down. The assumption is that a connection will be used for a stream of information so long that the overhead needed to set it up and tear it down are basically negligible. Most streams will consist of a file of one kind or another - a source program, a binary program, an input file, and so on. To see how long the average file is, we have conducted some measurements on the UNIX† system used in our department by the faculty and staff for research (no students, thus). The results of these measurements show that 34% of all files are less than 512 bytes, 52% are less than 1K bytes, 67% are less than 2K bytes, 79% are less than 4K bytes, 88% are less than 8K bytes, and 94% are less than 16K bytes.

The above considerations have led us to a different approach [Mullender83]. With packets of even 2K bytes, two thirds of all files fit into a single packet. Consequently, it is much simpler to adopt a "Request-Reply" or "Transaction" style of communication, in which the basic primitive is the client sending a request to a server and the server sending a reply back to the client. The client uses trans and the server getreq and putrep. Trans sends a request, and blocks until a reply is received. Getreq blocks the server until a request is received, which can then be processed, after which a reply can be sent using putrep. Each request-reply pair is completely self-contained, and independent of any other ones that may previously been sent. In other words, no concept of a "connection" exists. Not only is this conceptually much more appropriate

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

for use in an operating system, but it is much simpler to implement than a complex 7-layer protocol, not to mention offering lower delay.

As a matter of fact, a distinct trend towards connectionless interprocess communication services could clearly be observed at the recent Workshop on Operating Systems in Computer Networks in Zürich, Switzerland: all, or nearly all of the systems presented there were message-based rather than connection-based.

Henceforth we will refer to a request-reply pair as a *transaction*, which is not to be confused with transactions with a data base.

2.2. Basic communication protocol

Instead of a 7-layer protocol, we effectively have a 4-layer protocol. The bottom layer is the Physical Layer, and deals with the electrical, mechanical and similar aspects of the network hardware. The next layer is the Port Layer, and deals with the location of services, the transport of (32K byte) datagrams (packets whose delivery is not guaranteed) from source to destination and enforces the protection mechanism, which will be discussed in the next section. On top of this we have a layer that deals with the reliable transport of bounded length (32K byte) requests and replies between client and server. We have called this layer the Transaction Layer. The final layer has to do with the semantics of the requests and replies, for example, given that one can talk to the file server, what commands does it understand. The bottom three layers (Physical, Port and Transaction) are implemented by the kernel and hardware; only the Transaction Layer interface is visible to users.

Since systems of the kind we are describing will use high-speed, highly reliable local networks, few, if any, of the complex mechanisms designed for flow- and error-control in long-haul networks are useful here. Among other things, a simple stop-and-wait protocol is sufficient. The main function of the Transaction Layer is to provide an end-to-end *message* service built on top of the underlying *datagram* service, the main difference being that the former uses timers and acknowledgements to guarantee delivery whereas the latter does not.

The Transaction Layer protocol is straightforward. When the client does a trans, a packet, or sequence of packets, containing the request is sent to the server, the client is blocked, and a timer is started (inside the Transaction Layer). If the server does not acknowledge receipt of the request packet before the timer expires (usually by sending the reply, but in some special cases by sending a separate acknowledgement packet), the Transaction Layer retransmits the packet again and restarts the timer. When the reply finally comes in, the client sends back an acknowledgement (possibly piggybacked onto the next request packet) to allow the server to release any resources, such as buffers, that were acquired for this transaction. Under normal circumstances, reading a long file, for example, consists of the sequence

From client : request for block 0
From server: here is block 0

From client : acknowledgement for block 0 and request for block 1
From server: here is block 1
etc.

The protocol can handle the situation of a server crashing and being rebooted quite easily since each request contains the capability for the file to be read and the position in the file to start reading. Between requests, the server has no "activation record" or other table entry whose loss during a crash causes the server to forget which files were open, etc., because no concept of an open file or a current position in a file exists on the server's side. Each new request is completely self-contained. Of course for efficiency reasons, a server may keep a cache of frequently accessed i-nodes, file blocks etc., but these are not essential and their loss during a crash will merely slow the server down slightly while they are being dynamically refreshed after a reboot.

2.3. The port layer

The Port Layer is responsible for the speedy transmission of 32K byte datagrams. The Port Layer need only do this reasonably reliably, and does not have to make an effort to guarantee the correct delivery of every datagram. This is the responsibility of the Transaction Layer. Our results show that this approach leads to significantly higher transmission speeds, due to simpler protocols.

Theoretically, very high speeds are achievable in modern local-area networks. A typical speed for DMA transfers is 1 byte/μsec, and the typical transmission speed of a 10 Mbit local-area network is also 1 byte/μsec. Since, in many network interfaces, DMA transfer and network transfer cannot overlap, but DMA at the destination host *can* overlap with the DMA of the next packet at the source host, an upper bound for the transfer rate of a typical local-area network is 500,000 bytes/sec point-to-point.

In practise, however, speeds of 100,000 bytes per second between user processes have rarely been achieved. Obviously, to achieve higher transmission rates, the overhead of the protocol must be kept very low indeed, while an effort must be made to overlap DMA s at both communicating parties. To achieve this, we have chosen a large datagram size for the Port Layer, which has to split up the datagrams into small packets that the network hardware can cope with. This approach allows the implementor of the Port Layer to exploit the possibilities that the hardware has to offer to achieve an efficient stream of packets.

Our implementation of the Port Layer interfaces to a 10 Mbit token ring that allows *scatter-gather ;* that is, a packet can be sent to or from the interface in several DMA transfers, and then transmitted over the network separately. This allows us to do two important things to speed up the protocol. First, when a packet is received, the header can be inspected separately, so the protocol can decide where in memory the packet must go. The protocol driver can then transfer the packet directly from the interface to the right place in memory, without having to copy it. A copy loop would halve the transmission

speed. Second, the separation of DMA and transmission allows the driver to prepare a transmission by doing the DMA. The transmission can then be initiated immediately when the signal is received that the receiver is ready. In our implementation of the Port Layer, these considerations have resulted in the protocol that will now be described.

The transmitter begins by transferring and sending the first 2K of the datagram to be transmitted (2K is the maximum packet size allowed by the hardware). Immediately after the transmission is complete, the DMA for the next 2K bytes is started, but they are not yet transmitted. In the mean time, the receiver is interrupted by the arrival of the first packet. It extracts the header, examines it and decides where the body of the packet should go. Then the body of the packet is transferred from the interface to its final location in memory. While this is being done, the receiver prepares a tiny *acknowledgement* packet to tell the transmitter it is prepared for the next packet. As soon as the DMA transfer of the previous packet has finished, this acknowledgement is sent back to the transmitter. When the transmitter receives it, the transfer of the next packet to the interface will have finished, so it can then be sent immediately. This sequence is continued until the whole datagram is transmitted.

2.4. The transaction layer

It is the responsibility of the Transaction Layer to guarantee the arrival of requests and replies. The Transaction Layer makes use of the Port Layer and timers to achieve this.

The interface to the transaction layer basically consists of three calls, one for clients, and two for servers. All calls use a small datastructure, called Mref, which contains a pointer to a small fixed-size out-of-band buffer for the transmission of commands and parameters to the server, a pointer to the main body of data to be transferred, and the length of the main body of data (0 to 32768), as follows:

```
typedef struct Mref {
        char    *M_oob;
        char    *M_buf;
        unsigned M_len;
} Mref;

typedef struct Cap {
        Port    C_port;         /* 6-byte port */
        char    C_private[10];   /* 10-byte private */
} Cap;   /* capability */
```

The client, in order to do a transaction calls

```
trans(cap, req, rep);
        Cap *cap;
        Mref *req, *rep;
```

The server receives requests and sends replies with

```
getreq(port, cap, req);
        Port *port;
        Cap *cap;
        Mref *req;

putrep(rep);
        Mref *rep;
```

In principle, the Transaction Layer works as follows: When a client calls **trans**, the Transaction Layer generates a *reply-port* to enable the server to send a reply. The server port is deduced from the capability; the first 48 bits of the capability for an object identify the service that controls the object. The request is then sent, using **put**, and a *retransmission timer* is started.

The server, which previously had made a call to **getreq**, receives the request; the capability is filled in, and the received message is put in the buffers referred to by **req**. As soon as the request is received, the server's Transaction Layer starts a *piggyback timer*. When the server has not sent a reply before this timer expires, a separate acknowledgement is sent to put the client at ease, and stop its retransmission timer. When the server sends a reply to the client the same thing happens, more or less, with the role of client and server reversed. When a client makes a sequence of transactions with a single server, a subsequent request will acknowledge receipt of the previous reply.

The client maintains one more timer, the *crash timer*. This timer is set when the server's acknowledgement to a request has been received, and is used to detect server crashes. Whenever this timer expires, the client sends an "are you still alive?" packet to the server, to which the server replies with an acknowledgement.

When transactions occur quickly, one after the other, no extra acknowledgements are sent at all. Only when transactions take a long time (say, longer than a minute), acknowledgements are sent, and when transactions take much longer than that (say, ten minutes) then "are you still alive" messages begin to be sent.

2.5. Timer management

If the timers are started and stopped in exactly the way described above, the Transaction Layer would become unacceptably slow. Per (quick) transaction, two retransmission timers and two piggyback timers would have to be started and stopped, eight timer actions altogether.

There is a much more efficient way of dealing with timers, one that makes use of a *sweep algorithm*. This algorithm does not implement very accurate timers, but accuracy of the timer intervals is not very important to the correct

and efficient operation of the protocol.

The sweep algorithm is run every N clock ticks. N must be chosen such that N ticks is about the minimum timer interval needed (the piggyback timer interval). Whenever the algorithm is called, it makes a sweep over all outstanding transactions. If the state of a transaction has changed, the new state is recorded. If it has not changed, a counter is incremented, telling for how long the state has remained the same. If the (state, counter) combination has reached a certain value, the sweep algorithm carries out the appropriate actions, usually sending an acknowledgement, retransmitting a message, or aborting a transaction.

Because this algorithm is used there is no code needed in the transaction code itself, reducing the overhead of the Transaction Layer significantly. In this way, the code executed in the Transaction Layer is optimised for the normal case (no errors).

2.6. Blocking vs. non-blocking transaction primitives

Most services need to be able to handle multiple requests from different clients simultaneously. It therefore seems natural to implement non-blocking calls for interprocess communication, as this will allow a service to react to events in the order they occur. When blocking communication calls are used, a server is forced to wait for the specific event that unblocks the call.

Because it is rather difficult to write correct code for a process which has to handle multiple flows of control indeterministically, the ***Amoeba*** system provides the concept of *tasks*, sharing an address space. A number of tasks in one address space forms a *cluster*, and specific rules govern the scheduling of tasks within a cluster: only one task can run at a time, and a task runs until it voluntarily relinquishes control (e.g., on `trans` and `getreq` calls).

A server can thus easily be structured as a collection of co-operating tasks, each task handling one request. This model has greatly simplified the structure of services, as each task making up the server cluster now has a single thread of execution. The model also obviated the need for non-blocking transaction calls, with their complicated (and slow) extra interface for handling interrupts.

2.7. Results

Two versions of the algorithm have now been implemented. The one described has been implemented on the ***Amoeba*** distributed operating system, and achieves over 300,000 bytes a second from user process to user process (using M68000s and a Pronet* ring). It is now being implemented under UNIX where we expect to obtain more than 200,000 bytes/sec, assuming the communicating processes are not swapped.

An older version of the protocol, using 2K byte datagrams, now gets 90,000 bytes/sec across the network between two VAX-750s running a normal load of work, without causing a significant load on the system itself.

* PRONET is a trademark of Proteon Associates, Inc.

Several services, implemented under UNIX, are using the Transaction Layer interface, and it is our experience that these services are easy to design and that they work efficiently.

3. Ports and capabilities

3.1. Ports

Every service has one or more *ports* [Mullender84] to which client processes can send messages to contact the service. Ports consist of large numbers, typically 48 bits, which are known only to the server processes that comprise the service, and to the service's clients. For a public service, such as the system file service, the port will be generally made known to all users. The ports used by an ordinary user process will, in general, be kept secret. Knowledge of a port is taken by the system as prima facie evidence that the sender has a right to communicate with the service. Of course the service is not required to carry out work for clients just because they know the port, for example, the public file service may refuse to read or write files for clients lacking account numbers, appropriate authorization, etc.

Although the port mechanism provides a convenient way to provide partial authentication of clients ("if you know the port, you may at least talk to the service"), it does not deal with the authentication of servers. The basic primitive operations offered by the system are `trans`, `putreq` and `getrep`. Since everyone knows the port of the file server, as an example, how does one insure that malicious users do not execute `getreq`s on the file server's port, in effect impersonating the file server to the rest of the system?

One approach is to have all ports manipulated by kernels that are presumed trustworthy and are supposed to know who may `getreq` from which port [Cheriton83, Rashid81]. We reject this strategy because some machines, e.g., personal computers connected to larger multimodule systems may not be trustworthy, and also because we believe that by making the kernel as small as possible, we can enhance the reliability of the system as a whole. Instead, we have chosen a different solution that can be implemented in either hardware or software. First we will describe the hardware solution; later we will describe the software solution.

In the hardware solution, we need to place a small interface box, which we call an F-box (Function-box) between each processor module and the network. The most logical place to put it is on the VLSI chip that is used to interface to the network. Alternatively, it can be put on a small printed circuit board inside the wall socket through which personal computers attach to the network. In those cases where the processors have user mode and kernel mode and a trusted operating system running in kernel mode, it can also be put into operating system software. In any event, we assume that somehow or other all packets entering and leaving every processor undergo a simple transformation that users cannot bypass.

The transformation works like this. Each port is really a pair of ports, P, and G, related by: $P = F(G)$, where F is a (publicly-known) one-way function

[Wilkes68, Purdy74, Evans74] performed by the F-box. The one-way function has the property that given G it is a straightforward computation to find P, but that given P, finding G is so difficult that the only approach is to try every possible G to see which one produces P. If P and G contain sufficient bits, this approach can be made to take millions of years on the world's largest supercomputer, thus making it effectively impossible to find G given only P. Note that a one-way function differs from a cryptographic transformation in the sense that the latter must have an inverse to be useful, but the former has been carefully chosen so that no inverse can be found.

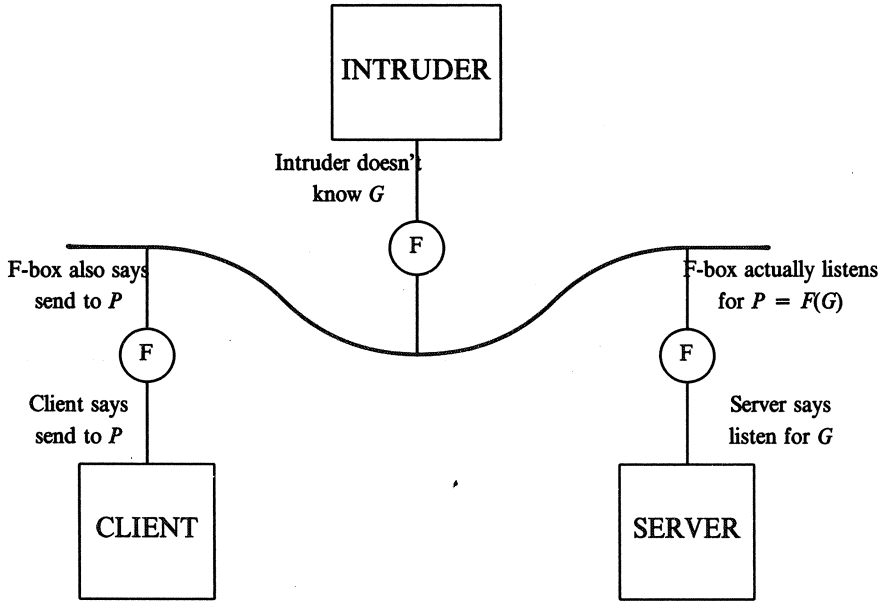


FIGURE 1.

Using the one-way F-box, the server authentication can be handled in a simple way, illustrated in figure 1. Each server chooses a get-port, G, and computes the corresponding put-port, P. The get-port is kept secret; the put-port is distributed to potential clients or in the case of public servers, is published. When the server is ready to accept client requests, it does a `getreq(`G`, cap, req)`. The F-box then computes $P = F(G)$ and waits for packets containing P to arrive. When one arrives, it is given to the appropriate process. To send a packet to the server, the client merely does `trans(cap, req, rep)`, where the *port* field of `cap` is set to P. This will cause a datagram to be sent by the local F-box with P in the destination-port field of the header. The F-box on the sender's side does not perform any transformation on the P field of the outgoing packet.

Now let us consider the system from an intruder's point of view. To impersonate a server, the intruder must do `getreq(`G`, ··· )`. However, G is a well-kept secret, and is never transmitted on the network, Since we have

assumed that G cannot be deduced from P (the one-way property of F) and that the intruder cannot circumvent the F-box, he cannot intercept packets not intended for him. Replies from the server to the client are protected the same way, only with the client's Transaction Layer picking a get-port for the reply, say, G', and including $P' = F(G')$ in the request packet.

The presence of the F-box makes it easy to implement digital signatures for still further authentication, if that is desired. To do so, each client chooses a random signature, S, and publishes $F(S)$. The F-box must be designed to work as follows. Each packet presented to the F-box contains three special header fields: destination (P), reply (G'), and signature (S). The F-box applies the one-way function to the second and third of these, transmitting the three ports as: P, $F(G')$, and $F(S)$, respectively. The first is used by the receiver's F-box to admit only packets for which the corresponding `getreq` has been done, the second is used as the put-port for the reply, and the third can be used to authenticate the sender, since only the true owner of the signature will know what number to put in the third field to insure that the publicly-known $F(S)$ comes out.

It is important to note that the F-box arrangement merely provides a simple *mechanism* for implementing security and protection, but gives operating system designers considerable latitude for choosing various *policies*. The mechanism is sufficiently flexible and general that it should be possible to put it into hardware with precluding many as-yet-unthought-of operating systems to be designed in the future.

3.2. Capabilities

In any object-based system, a mechanism is needed to keep track of which processes may access which objects and in what way. The normal way is to associate a capability with each object, with bits in the capability indicating which operations the holder of the capability may perform. In a distributed system this mechanism should itself be distributed, that is, not centralized in a single monolithic "capability manager." In our proposed scheme, each object is managed by some service, which is a user (as opposed to kernel) program, and which understands the capabilities for its objects.

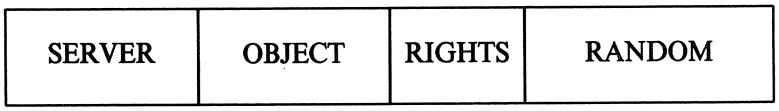


FIGURE 2.

A capability typically consists of four fields, as illustrated in figure 2:

1. The put-port of the service that manages the object
2. An Object Number meaningful only to the service managing the object
3. A Rights Field, which contains a 1 bit for each permitted operation
4. A Random Number for protecting each object

The basic model of how capabilities are used can be illustrated by a simple

example: a client wishes to create a file using the file service, write some data into the file, and then give another client permission to read (but not modify) the file just written. To start with, the client sends a message to the file service's put-port specifying that a file is to be created. The request might contain a file name, account number and similar attributes, depending on the exact nature of the file service. The server would then pick a random number, store this number in its object table, and insert it into the newly-formed object capability. The reply would contain this capability for the newly created (empty) file.

To write the file, the client would send a message containing the capability and some data. When the `write` request arrived at the file server process, the server would normally use the object number contained in the capability as as index into its tables to locate the object. For a UNIX like file server, the object number would be the i-node number, which could be used to locate the i-node.

Several object protection systems are possible using this framework. In the simplest one, the server merely compares the random number in the file table (put there by the server when the object was created) to the one contained in the capability. If they agree, the capability is assumed to be genuine, and all operations on the file are allowed. This system is easy to implement, but does not distinguish between `read`, `write`, `delete`, and other operations that may be performed on objects.

However, it can easily be modified to provide that distinction. In the modified version, when a file (object) is created, the random number chosen and stored in the file table is used as an encryption/decryption key. The capability is built up by taking the Rights Field (e.g., 8 bits), which is initially all 1s indicating that all operations are legal, and the Random Number Field (e.g., 56 bits), which contains a known constant, say, 0, and treating them as a single number. This number is then encrypted by the key just stored in the file table, and the result put into the newly minted capability in the combined Rights-Random Field. When the capability is returned for use, the server uses the object number (not encrypted) to find the file table and hence the encryption/decryption key. If the result of decrypting the capability leads to the known constant in the Random Number Field, the capability is almost assuredly valid, and the Rights Field can be believed. Clearly, an encryption function that mixes the bits thoroughly is required to ensure that tampering with the Rights Field also affects the known constant. Exclusive or'ing a constant with the concatenated Rights and Random fields will not do.

When this modified protection system is used, the owner of the object can easily give an exact copy of the capability to another process by just sending it the bit pattern, but to pass, say, read-only access, is harder. To accomplish this task, the process must send the capability back to the server along with a bit mask and a request to fabricate a new capability whose Rights Field is the Boolean-and of the Rights Field in the capability and the bit mask. By choosing the bit mask carefully, the capability owner can mask out any operations that the recipient is not permitted to carry out.

This modified system works well except that it requires going back to the server every time a sub-capability with fewer rights is needed. We have devised yet another protection system that does not have this drawback. This third scheme requires the use of a set of N commutative one-way functions, $F_0, F_1, \cdots, F_{N-1}$ corresponding to the N rights present in the Rights Field. When an object is created, the server chooses a random number and puts it in both the file table and the Random Number Field, just as in the first scheme presented. It also sets all the Rights Field bits to 1.

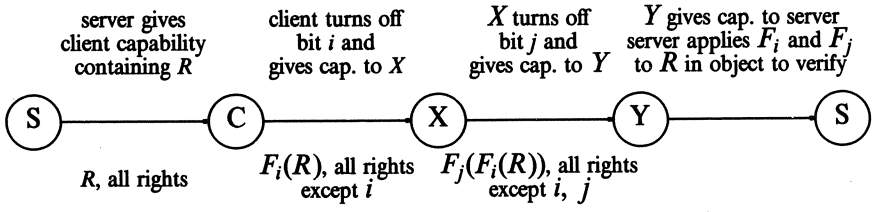


FIGURE 3.

A client can delete permission k from a capability by replacing the random number, R, with $F_k(R)$ and turning off the corresponding bit in the Rights Field. When a capability comes into the server to be used, the server fetches the original random number from the file table, looks at the Rights Field, and applies the functions corresponding to the deleted rights to it. If the result agrees with the number present in the capability, then the capability is accepted as genuine, otherwise it is rejected. The mechanism is illustrated in figure 3. Note that although the Rights Field is not encrypted, it is pointless for a client to tamper with it, since the server will detect that immediately. In theory at least, the Rights Field is not even needed, since the server could try all 2^N combinations of the functions to see if any worked. Its presence merely speeds up the checking. It should also be clear why the functions must be commutative - it does not matter in what order the bits in the Rights Field were turned off.

The organization of capabilities and objects discussed above has the interesting property that although no central record is kept of who has which capabilities, it is easy to retract existing capabilities. All that the owner of an object need do is ask the server to change the random number stored in the file table. Obviously this operation must be protected with a bit in the Rights Field, but if it succeeds, all existing capabilities are instantly invalidated.

3.3. Protection without F-boxes

Earlier we said that protection could also be achieved without F-boxes. It is slightly more complicated, since it uses both conventional and public-key encryption, but it is still quite usable. The basic idea underlying the method is the fact that in nearly all networks an intruder can forge nearly all parts of a packet being sent except the source address, which is supplied by the network

interface hardware. To take advantage of this property, imagine a (possibly symmetric) conceptual matrix of conventional (e.g., DES) encryption keys, with the rows being labeled by source machine and the columns by destination machine. Thus the matrix selects a unique key for encrypting the *capabilities* in any packet. The data need not be encrypted, although that is also possible if needed.

Each machine is assumed to know its row and column of the matrix, and nothing else (how this will be achieved will be discussed shortly). With this arrangement, intruder I can easily capture packets from client C to server S, but attempts to "play them back" to the server will fail because the server will see the source machine as I (assumed unforgeable) and use element M_{IS} as the decryption key instead of the correct M_{CS}. No matter what the intruder does, he cannot trick the server into using a decryption key that decrypts the capabilities to make sense, that is, to contain random numbers that agree with those stored in the file tables.

To avoid having to run the encryption/decryption algorithm frequently, all machines can maintain a hashed cache of capabilities that they have been using frequently. Clients will hash their caches on the unencrypted capabilities in the form of triples: (unencrypted capability, destination, encrypted capability), whereas servers will hash theirs in the form of triples: (encrypted capability, source, unencrypted capability).

To set up the matrix initially, the following procedure can be used. A public server, such as a file server, makes its put-port and a public encryption key known to the whole world. When a new machine joins the network (e.g., after a crash or upon initial system boot), it sends a broadcast message announcing its presence. Suppose, for example, the file server has just come up, and must (1) prove that it is the file server to other processes, and (2) establish the conventional keys used for encrypting capabilities in both directions.

A client machine, C, which receives the broadcast from the alleged file server, F, picks a new conventional encryption key, K, for use in subsequent C to F traffic and sends it to F encrypted with F's public key. F then decrypts K and replies to C by sending a packet containing both K and a newly chosen conventional key to be used for reverse traffic. This packet is encrypted both with K itself and with the inverse of F's public key, so C can use K and F's public key to decrypt it. If the decrypted packet contains K, C can be sure that the other conventional key was indeed generated by the owner of F's public key, thus convincing C that he is indeed talking to the file server. Both of the above-mentioned conditions have now been fulfilled, so normal communication can now take place. Note that the use of different conventional keys after each reboot make it impossible for an intruder to fool anyone by playing back old packets.

4. The Amoeba file system

The file system has been designed to be highly modular, both to enhance reliability and to provide a convenient testbed for doing research on distributed file systems. It consists of three completely independent pieces: the block service, the file service, and the directory service. In short, the block service provides commands to read and write raw disk blocks. As far as it is concerned, no two blocks are related in any way, that is, it has no concept of a file or other aggregation of blocks. The file service uses the block service to build up files with various properties. Finally, the directory service provides a mapping of symbolic names onto object capabilities.

4.1. Block service

The block service is responsible for managing raw disk storage. It provides an object-oriented interface to the outside world to relieve file servers from having to understand the details of how disks work. The principle operations it performs are:

- `allocate` a block, write data into it, and return a capability to the block
- given a capability for a block, `free` the block
- given a capability for a block, `read` and return the data contained in it
- given a capability for a block and some data, `write` the data into the block
- given a capability for a block and a key, `lock` or unlock the block

These primitives provide a convenient object-oriented interface for file servers to use. In fact, any client who is unsatisfied [Stonebraker81, Tanenbaum82] with the standard file system can use these operations to construct his own.

The first four operations of `allocate`, `free`, `read`, and `write` hardly need much comment. The fifth one provides a way for clients to lock individual blocks. Although this mechanism is crude, it forms a sufficient basis for clients (e.g., file systems) to construct more elaborate locking schemes, should they so desire.

One other operation is worth noting. The data within a block is entirely under the control of the processes possessing capabilities for it, but we expect that most file servers will use a small portion of the data for redundancy purposes. For example, a file server might use the first 32 bits of data to contain a file number, and the next 32 bits to contain a relative block number within the file. The block server supports an operation `recovery`, in which the client provides the account number it uses in `allocate` operations and requests a list of all capabilities on the whole disk containing this account number. (The block server stores the account number for each block in a place not accessible to clients.) Although `recovery` is a very expensive operation, in effect requiring a search of the entire disk, armed with all the capabilities returned, a file server that lost all of its internal tables in a crash could use the first 64 bits of each block to rebuild its entire file list from scratch.

4.2. File service

The purpose of splitting the block service and file service is to make it easy to provide a multiplicity of different file services for different applications. One such file service that we envision is one that supports flat files with no locking, in other words, the UNIX model of a file as a linear sequence of bytes with no internal structure and essentially no concurrency control. This model is quite straightforward and will therefore not be discussed here further.

A more elaborate file service with explicit version and concurrency control for a multiuser environment will be described instead [Mullender85]. This file service is designed to support data base services, but it itself is just an ordinary, albeit slightly advanced, file service. The basic model behind this file service is that a file is a time-ordered sequence of versions, each version being a snapshot of the file made at a moment determined by a client [Fridrich81, Reed81]. At any instant, exactly one version of the file is the *current version*. To use a file, a client sends a message to a file server process containing a file capability and a request to create a new, private version of the current version. The server returns a capability for this new version, which acts like it is a block for block copy of the current version made at the instant of creation. In other words, no matter what other changes may happen to the file while the client is using his private version, none of them are visible to him. Only changes he makes himself are visible.

Of course, for implementation efficiency, the file is not really copied block for block. What actually happens is that when a version is created, a table of pointers (capabilities) to all the file's blocks is created. The capability granted to the client for the new version actually refers to this version table rather than the file itself. Whenever the client reads a block from the file, a bit is set in the version table to indicate that the corresponding block has been read. When a block is modified in the version, a new block is allocated using the block server, the new block replaces the original one, and its capability is inserted into the version table. A bit indicating that the block is a new one rather than an original is also set. This mechanism is sometimes called "copy on write."

Versions that have been created and modified by a client are called *uncommitted versions*. At a particular moment, the current version may have several (different) uncommitted versions derived from it in use by different clients. When a client is finished modifying his private version, he can ask the file server to *commit* his version, that is, make it the current version instead of the then current version. If the version from which the to-be-committed version was derived is still current at the time of the commit, the commit succeeds and becomes the new current version.

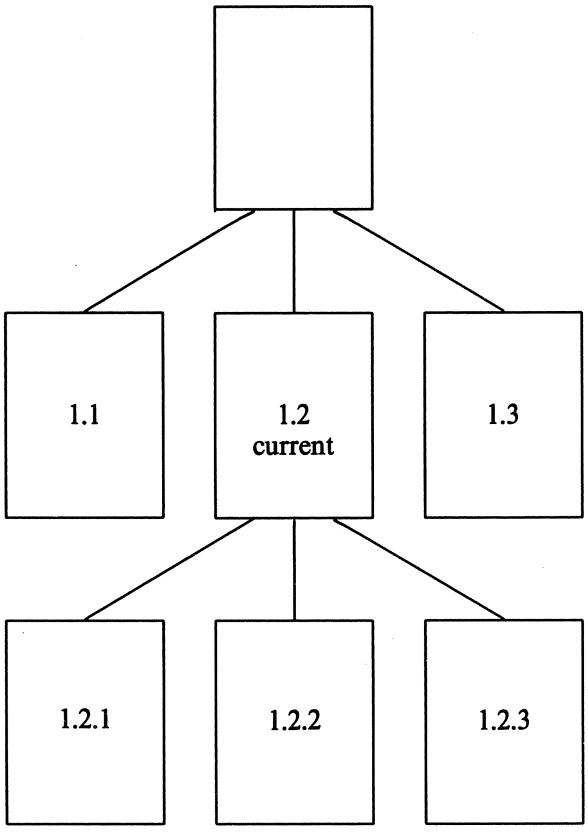


FIGURE 4.

As an example, suppose version 1 is initially the current version, with various clients creating private versions 1.1, 1.2, and 1.3 based on it. If version 1.2 is the first to commit, it wins and 1.2 becomes the new current version, as illustrated in figure 4. Subsequent requests by other clients to create a version will result in versions 1.2.1, 1.2.2, and 1.2.3, all initially copies of 1.2.

The fun begins when the owner of version 1.3 now tries to commit. Version 1, on which it is based, is no longer the current version, so a problem arises. To see how this should be handled, we must introduce a concept from the data base world, *serializability* [Eswaran76, Papadimitriou79]. Two updates to a file are said to be serializable if the net result is either the same as if they were run sequentially in either order. As a simple example, consider a two character file initially containing "ab." Client 1 wants to write a "c" into the first character, wait a while, and then write a "d" into the second character. Client 2 wants to write an "e" into the first character, wait a while, and then write an "f" into the second character. If 1 runs first we get "cd"; if 2 runs first we get "ef." Both of these are legal results, since the file server cannot dictate when the users run. However, its job is to prevent final configurations of "cf" or "de,"

both of which result from interleaving the requests. If a client locks the file before starting, does all its work, and then unlocks the file, the result will always be either "cd" or "ef," but never "cf" or "de." What we are trying to do is accomplish the same goal without using locking.

The idea behind not locking is that most updates, even on the same file, do not affect the same parts of the file, and hence do not conflict. For example, changes to an airline reservation data base for flights from San Francisco to Los Angeles do not conflict with changes for flights from Amsterdam to London. The strategy behind our commit mechanism is to let everyone make and modify versions at will, with a check for serializability when a commit is attempted. This mechanism has been proposed for data base systems [Kung81], but as far as we know, not for file systems.

The serializability check is straightforward. If a version to be committed, *A*, is based on the version that is still current, *B*, it is serializable and the commit succeeds. If it is not, a check must be made to see if all of the blocks belonging to *A* that the client has read are the same in the current version as they were in the version from which *A* was derived. If so, the previous commit or commits only changed blocks that the client trying to commit *A* was not using, so there is no problem and the commit can succeed.

If, however, some blocks have been changed, modifications that *A*'s owner has made may be based on data that are now obsolete, so the commit must be refused, but a list is returned to *A*'s owner of blocks that caused conflicts, that is, blocks marked "read" in *A* and marked "written" in the current version (or any of its ancestors up to the version on which *A* is based). At this point, *A*'s owner can make a new version and start all over again. Our assumption is that this event is very unlikely, and that is occasional occurrence is a price worth paying for not having locking, deadlocks, and the delays associated with waiting for locks.

4.3. Directory service

Because it is frequently inconvenient to deal with long binary bit strings such as capabilities, a directory service is needed to provide symbolic naming. The directory service's task is to manage directories, each of which contains a collection of (ASCII name, capability) pairs. The principal operation on a directory object is for a client to present a capability for a directory and an ASCII name, and request the directory service to look up and return the capability associated with the ASCII name. The inverse operation is to store an (ASCII name, capability) pair in a directory whose capability is presented.

5. PROCESS MANAGEMENT

Like any other operating system, this one must also have a way to manage processes. In our design, processes are created and managed by the process service, which consists of three major subsystems, the generic server, the process server, and the boot server.

5.1. Generic server

The idea behind the generic server is that much of the time a user wants a certain program to be run, but does not care about where it is run or on which CPU type. For example, a user might have a Pascal program to be compiled, and wants a Pascal compiler that produces, say, Motorola 68000 code. However, he does not care whether the compiler itself runs on a 68000, a VAX or any other CPU. We speak of this as a generic Pascal compiler.

The generic server's job is to locate a suitable hardware/software combination and start it up. This can be done by maintaining internal tables of locations where the appropriate service is likely to be located. By sending a message to the chosen service, the generic server can see if the corresponding server is currently available and willing to take on the offered work. If so, it can begin; if not, the generic server can broadcast a request for bids to see if someone else can be located. If no willing server exists, the generic server will have to cause one to be created by invoking the process server.

5.2. Process Server

The process server's job is to take a process descriptor sent to it, locate a free processor, and send sufficient information to the processor to allow the processor to run. The process descriptor must contain at least the following information:

1. The CPU type desired.
2. A capability for the binary file to be executed.
3. Capabilities for process environment.
4. Accounting information.

The CPU type and binary file capability are obvious. The third item has to do with things like the file descriptors and environment strings in UNIX. When a UNIX process is started up, it inherits certain parameters from its parent, among these are usually file descriptors for standard input, output, and diagnostic, and possibly other files as well. In our design, a process can inherit capabilities for standard input, standard output, and standard diagnostic, as well as other ones. By using these, one can implement UNIX pipes and filters easily, as well as more general mechanisms (e.g., passing capabilities to third parties, storing them in files for later use, etc.).

Another area that the process service must deal with is scheduling. It must allocate processes to processors, and possibly control migration and swapping among processors as well. By introducing the concept of a "process image" which contains all the information necessary to run a process (e.g., its memory, registers, capabilities, etc.) it becomes straightforward to handle process migration and swapping in a unified way. When a process is swapped out to a disk somewhere, there is no need to have it swapped back to the same machine that it originated on.

5.3. Boot service

Many services must achieve high availability. Our approach to this issue is using fault tolerance, rather than fault intolerance. In the former, one expects hardware and software to fail, and makes provision for dealing with it; in the latter, one assumes that they are perfect and that no such provision need be made. Since many services are faced with the same problem: how to provide high availability in the face of occasional crashes, we have abstracted out a common part of the crash recovery mechanism and put it into a separate service, the boot service.

Any service that wants to provide a continuous availability can register with the boot service. Such registration entails providing a polling message to send the service periodically, the expect reply, the polling frequency, and a prescription of what to do in case of failure. The boot service then sends the polling message to the service at the requested frequency. As long as the service continues to send the appropriate reply, all is well and the boot service has nothing else to do.

However, if the service fails to reply properly, or fails to reply at all within an agreed upon time interval, the boot service declares the service to be out-of-order, and goes to the process service to start up a new version of it. Of course, the boot service itself must not crash, but it consists of a number of server processes that constantly check each other, and if need be, replace sick members with healthy ones.

6. Resource management

In keeping with our general philosophy of making the system kernel as small as possible, we have devised a way to put the resource control and accounting outside the kernel. Furthermore, a clear distinction is made between policy and mechanism, so that subsystem designers can implement their own policies with the standard mechanisms.

Traditionally, accounting was used by the management of a computer center to levy charges for the use of the computer center's resources: CPU time, file space, lineprinter paper. This method worked quite well in the past, when hardware resources were expensive compared to the software used. Nowadays, hardware is cheap, software expensive. However, in the traditional approach there is usually no possibility to bill users for the use of a particular piece of software, or to have one user bill another for using his services.

Additionally, distributed systems need not be under control of one centralized management any more; private, personal computers can be plugged into the network and both use and offer services to the rest of the network. The accounting mechanisms in a distributed systems must be able to handle this new view on operating systems and allow any user that sets up a service to gather information about who uses his service.

6.1. Bank service

The bank service is the heart of the resource management mechanism. It implements an object called a "bank account" with operations to transfer virtual money between accounts and to inspect the status of accounts. Bank accounts come in two varieties: individual and business. Most users of the system will just have one individual account containing all their virtual money. This money is used to pay for CPU time, disk blocks, typesetter pages, and all other resources for which the service owning the resource decides to levy a charge.

Business accounts are used by services to keep track of who has paid them and how much. Each business account has a subaccount for each registered client. When a client transfers money from his individual account to the service's business account, the money transferred is kept in the subaccount for that client, so the service can later ascertain each client's balance. As an example of how this mechanism works, a file service could charge for each disk block written, deducting some amount from the client's balance. When the balance reached zero, no more blocks could be written. Large advance payments and simple caching strategies can reduce the number of messages sent to a small number.

Another aspect of the bank service is its maintenance of multiple currencies. It can keep track of say, virtual dollars, virtual yen, virtual guilders and other virtual currencies, with or without the possibility of conversion among them. This feature makes it easy for subsystem designers to create new currencies and control how they are allocated among the subsystems users.

6.2. Accounting policies

The bank service described above allows different subsystems to have different accounting policies. For example, a file or block service could decide to use either a buy-sell or a rental model for accounting. In the former, whenever a block was allocated to a client, the client's account with the service would be debited by the cost of one block. When the block was freed, the account would be credited. This scheme provides a way to implement absolute limits (quotas) on resource use. In the latter model, the client is charged for rental of blocks at a rate of X units per kiloblock-second or block-month or something else. In this model, virtual money is constantly flowing from the clients to the servers, in which case clients need some form of income to keep them going. The policy about how income is generated and dispensed is determined by the owner of the currency in question, and is outside the scope of the bank server.

SUMMARY

This paper has discussed a model for a fifth generation computer system architecture and its operating system. The operating system is based on the use of objects protected by sparse capabilities. An outline of some of the key services has been given, notably the block, file, directory, generic, process, boot and

bank services.

REFERENCES

[Ball79]
BALL, J. E., BURKE, E. J., GERTNER, I., LANTZ, K. A., and RASHID, R. F., "Perspectives on Message-Based Distributed Computing," *Proc. IEEE*, 1979.

[Cheriton83]
CHERITON, D. R. and ZWAENEPOEL, W., "The Distributed V Kernel and its Performance for Diskless Workstations," *Proc. Ninth ACM Symp. on Operating Systems Principles*, pp.128-140, October 1983.

[Dennis66]
DENNIS, J. B. and HORN, E. C. VAN, "Programming Semantics for Multiprogrammed Computation," *Comm. ACM*, vol. 9, no. 3, pp.143-155, March 1966.

[Eswaran76]
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*, vol. 19, no. 11, pp.624-633, November 1976.

[Evans74]
EVANS, A., KANTROWITZ, W., and WEISS, E., "A User Authentication Scheme Not Requiring Secrecy in the Computer," *Comm. ACM*, vol. 17, no. 8, pp.437-442, August 1974.

[Fridrich81]
FRIDRICH, M. and OLDER, W., "The Felix File Server," *Proc. Eighth Symp. Operating Syst. Prin.*, pp.37-44, 1981, ACM.

[Kung81]
KUNG, H. T. and ROBINSON, J. T., "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp.213-226, June 1981.

[Liskov74]
LISKOV, B. and ZILLES, S., "Programming with Abstract Data Types," *SIGPLAN Notices*, vol. 9, pp.50-59, April 1974.

[Mullender83]
MULLENDER, SAPE J., RENESSE, ROBBERT VAN, and TANENBAUM, ANDREW S., "A Transaction-Oriented Transport Protocol", internal paper, Centre for Mathematics and Computer Science, 1983.

[Mullender84]
MULLENDER, S. J. and TANENBAUM, A. S., "Protection and Resource Control in Distributed Operating Systems," *Computer Networks*, vol. 8, no. 5,6, pp.421-432, 1984.

[Mullender85]
MULLENDER, S. J. and TANENBAUM, A. S., "A Distributed File Service Based on Optimistic Concurrency Control," *Proceedings of the 10th*

Symposium on Operating Systems Principles, pp.51-62, December 1985.

[Needham82]
NEEDHAM, R. M. and HERBERT, A. J., *The Cambridge Distributed Computer System.* Reading, Ma.: Addison-Wesley, 1982.

[Papadimitriou79]
PAPADIMITRIOU, C. H., "Serializability of Concurrent Updates," *J. ACM*, vol. 26, no. 4, pp.631-653, October 1979.

[Purdy74]
PURDY, G. B., "A High Security Log-in Procedure," *Comm. ACM*, vol. 17, no. 8, pp.442-445, August 1974.

[Rashid81]
RASHID, R., "Accent: A Network Operating System for SPICE/DSN", Tech. Rept., Computer Science Dept., Carnegie-Mellon University, May 1981.

[Reed81]
REED, D. and SVOBODOVA, L., "SWALLOW: A Distributed Data Storage System for a Local Network," *Proc. IFIP*, pp.355-373, 1981.

[Stonebraker81]
STONEBRAKER, M., "Operating System Support for Database Management," *Comm. ACM*, vol. 24, no. 7, pp.412-418, July 1981.

[Tanenbaum82]
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).

[Wilkes68]
WILKES, M. V., *Time-Sharing Computer Systems.* New York: American Elsevier, 1968.

[Zimmermann80]
ZIMMERMANN, H., "OSI Reference Model—The ISO Model of Architecture for Open Systems Intercon nection," *IEEE Trans. Comm.*, vol. COM-28, pp.425-432, April 1980.

Protection

Using Sparse Capabilities in a Distributed Operating System

Sape J. Mullender
Centre for Mathematics and Computer Science
Amsterdam, The Netherlands

Andrew S. Tanenbaum
Robbert van Renesse
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Most distributed operating systems constructed to date have lacked a unifying mechanism for naming and protection. In this paper we discuss a system, Amoeba, that uses capabilities for naming and protecting objects. In contrast to traditional, centralized operating systems, in which capabilities are managed by the operating system kernel, in Amoeba all the capabilities are managed directly by user code. To prevent tampering, the capabilities are protected cryptographically. The paper describes a variety of the issues involved, and gives four different ways of dealing with the access rights.

1. Introduction

Capabilities [Dennis and Van Horn 1966] have been used as the basis for a variety of uniprocessor operating systems (see [Levy 1984] for numerous examples). They have the attraction of providing a single, uniform mechanism for naming, accessing, and protecting all objects within the system. In all of these systems, the capabilities are managed by (trusted) kernel software, often with special assistance from the hardware.

The use of capabilities as a conceptual base for distributed systems has been minimal to date, a few exceptions being the Eden system [Almes et al. 1985], LINCS [Donnelley 1981], and ACCENT [Rashid 1981], Our scheme also uses a distributed capability mechanism, but it differs from each of these in significant ways, which we will describe after discussing our proposal.

This paper describes a scheme in which user processes manipulate

Using Sparse Capabilities in a Distributed Operating System
A. S. Tanenbaum, S. J. Mullender, and R. van Renesse
Proc. 6th Int. Conf. on Distributed Computing Systems
pp. 558-563
May 1986

capabilities directly in their own address spaces. Except for some very special parts of it, the kernel does not even know that capabilities are in use. To prevent users from forging new capabilities or tampering with existing ones, capabilities are protected cryptographically. This cryptographic protection scheme will first be described in some detail, followed by a discussion of how these capabilities are used in the Amoeba distributed operating system.

2. Ports and capabilities

2.1. Background on Amoeba

Amoeba is an object-oriented distributed operating system. Its semantic model is based on having client processes perform operations on objects managed by server processes. Objects are specified by capabilities. Operations are carried out by having processes exchange messages, generally in the form of a request from a client followed later by a reply from a server. The standard message format provides a place for one capability in the header, typically for the object being operated on, but users are free to put other capabilities in the data field as required. The header also contains room for the operation code and some parameters.

After making a request, a client blocks until the reply comes in, so the approach can be regarded as a simple remote procedure call mechanism [Spector 1982; Birrell and Nelson 1984]. The system does not use "connections" or virtual circuits or any other long-lived communication structures.

2.2. Ports

Every server has one or more *ports* to which client processes can send messages to contact the service (i.e., the server process). Ports consist of large numbers, typically 48 bits, which are known only to the server processes that comprise the service, and to the server's clients. For a public service, such as the file system, the port will generally be made known to all users. The ports used by an ordinary user process will, in general, be kept secret. Knowledge of a port is taken by the system as prima facie evidence that the sender has a right to communicate with the service. Of course the service is not required to carry out work for clients just because they know the port, for example, the file server will refuse to read or write files for clients lacking appropriate file capabilities. Thus two levels of protection are used here: ports for protecting access to servers, and capabilities for protecting access to individual objects. These two mechanisms are related, as will be shown later.

Although the port mechanism provides a convenient way to provide partial authentication of clients ("if you know the port, you may at least talk to the service"), it does not deal with the authentication of servers. How does one insure that malicious users do not listen on the file server's port, and try to impersonate the file server to the rest of the system?

One approach is to have all ports manipulated by kernels that are presumed trustworthy and are supposed to know who may listen on which port. As mentioned above, we reject this strategy because on some machines, e.g., per

so nal computers users may be able to tamper with the operating system kernel, and also because we believe that by making the kernel as small as possible, we can enhance the reliability of the system as a whole. Instead, we have chosen a different solution that can be implemented in either hardware or software.

In the hardware solution, we need to place a small interface box, which we call an F-box (Function-box) between each processor module and the network. The most logical place to put it is on the VLSI chip that is used to interface to the network. Alternatively, it can be put on a small printed circuit board inside the wall socket through which personal computers attach to the network. In those cases where the processors have user mode and kernel mode and the operating systems can be trusted, it could be put into operating system. In any event, we assume that somehow or other all messages entering and leaving every processor undergo a simple transformation that users cannot bypass.

The transformation works like this. Each port is really a pair of ports, P, and G, related by: $P = F(G)$, where F is a (publicly-known) one-way function [Wilkes 1968; Purdy 1974; Evans et al. 1974] performed by the F-box. The one-way function has the property that given G it is a straightforward computation to find P, but that given P, finding G is not feasible.

Using the one-way F-box, the server authentication can be handled in a simple way, as illustrated in figure 1. Each server chooses a get-port, G, and computes the corresponding put-port, P. The get-port is kept secret; the put-port is distributed to potential clients or in the case of public servers, is published. When the server is ready to accept client requests, it does a GET(G). The F-box then computes $P = F(G)$ and waits for messages containing P to arrive. When one arrives, it is given to the process that did GET(G). To send a message to the server, the client merely does PUT(P), which sends a message containing P in a header field to the server. The F-box on the sender's side does not perform any transformation on the P field of the outgoing message.

Now let us consider the system from an intruder's point of view. To impersonate a server, the intruder must do GET(G). However, G is a well-kept secret, and is never transmitted on the network. Since we have assumed that G cannot be deduced from P (the one-way property of F) and that the intruder cannot circumvent the F-box, he cannot intercept messages not intended for him. An intruder doing GET(P) will simply cause his F-box to listen to the (useless) port F(P). Replies from the server to the client are protected the same way, only with the client picking a get-port for the reply, say, G', and including $P' = F(G')$ in the request message.

The presence of the F-box makes it easy to implement digital signatures for still further authentication, if that is desired. To do so, each client chooses a random signature, S, and publishes $F(S)$. The F-box must be designed to work as follows. Each message presented to the F-box for transmission contains three special header fields: destination (P), reply (G'), and signature (S). The F-box applies the one-way function to the second and third of these, transmitting the three ports as: P, $F(G')$, and $F(S)$, respectively. The first is used by the receiver's F-box to admit only messages for which the

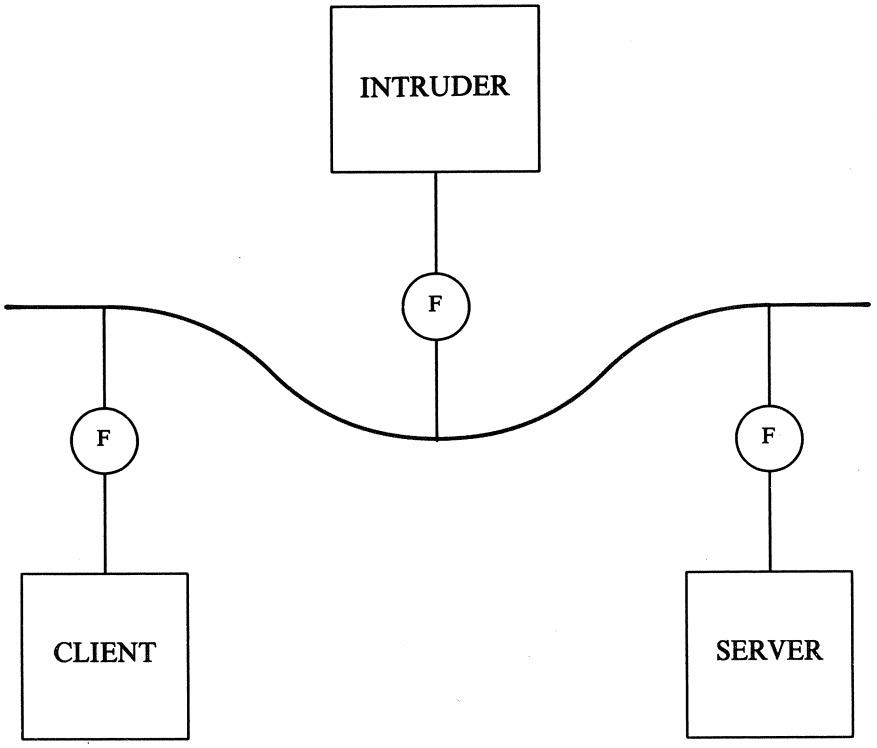


FIGURE 1. Clients, servers, intruders, and F-boxes.

corresponding GET has been done, the second is used as the put-port for the reply, and the third can be used to authenticate the sender, since only the true owner of the signature will know what number to put in the third field to insure that the publicly-known $F(S)$ comes out.

It is important to note that the F-box arrangement merely provides a simple *mechanism* for implementing security and protection, but gives operating system designers considerable latitude for choosing various *policies*. The mechanism is sufficiently flexible and general that it should be possible t o put it into hardware without precluding many as-yet-unthought-of operating systems to be designed in the future. In effect, it is a protected associative addressing scheme. The associative addressing can be simulated in software when the kernels are trusted by having each one maintain a cache of (port, machine-number) pairs. If a port is not in the cache, it can be found by broadcasting a LOCATE message. How this can be carried out efficiently, even in a network without broadcasting, is discussed in [MULLENDER and VITANYI 1984], along with many of the implications of location dependendent addressing, process migration, etc.

2.3. Capabilities

In any object-based system, a mechanism is needed to keep track of which processes may access which objects and in what way. The normal way is to associate a capability with each object, with bits in the capability indicating which operations the holder of the capability may perform. In a distributed system this mechanism should itself be distributed, that is, not centralized in a single monolithic "capability manager." In our proposed scheme, each object is managed by some server, which itself is a user (as opposed to kernel) process, and which understands the capabilities for its objects.

A capability typically consists of four fields as illustrated in figure 2.

1. The put-port of the server that manages the object
2. An object number meaningful only to the server managing the object
3. A rights field, containing a 1 bit for each permitted operation
4. A random number, for protecting each object

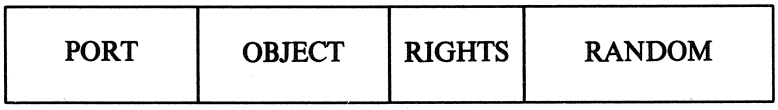


FIGURE 2. A Capability

The basic model of how capabilities are used and protected can be illustrated by a simple example: a client wishes to create a file using the file server, write some data into the file, and then give another client permission to read (but not modify) the file just written. To start with, the client sends a message to the file server's put-port specifying that a file is to be created. The request might contain a file name, account capability, etc. The server would then pick a random number, store this number in its object table, and insert it into the newly-formed object capability. The reply would contain this capability for the newly created (empty) file.

To write the file, the client would send a succession of data messages, each containing the capability and some data. When each WRITE request arrived at the file server process, the server would use the OBJECT field contained in the capability as as index into its file tables to locate the object. For a UNIX† like file server, the object number would be the i-number, which could be used to locate the i-node.

Several object protection systems are possible using this framework. In the simplest one, the server merely compares the random number in the file table (put there by the server when the object was created) to the one contained in the capability. If they agree, the capability is assumed to be genuine, and all operations on the file are allowed. This system is easy to implement, but does not distinguish between READ, WRITE, DELETE, and other operations that may be performed on objects.

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

However, the basic idea can easily be modified to provide that distinction. We will now describe three different algorithms for protecting the access rights. In the first version, when a file (object) is created, the random number chosen and stored in the file table is used as an encryption/decryption key. The capability is built up by taking the RIGHTS field, which is initially all 1s to indicate that all operations are legal, and the RANDOM field (e.g., 48 bits), which contains a known constant, say, 0, and treating them as a single number. This number is then encrypted by the key just stored in the file table, and the result put into the newly minted capability in the combined RIGHTS-RANDOM field.

When the capability is returned for use, the server uses the OBJECT field (not encrypted) to find the file table and hence the encryption/decryption key. If the result of decrypting the capability leads to the known constant in the RANDOM field, the capability is almost assuredly valid, and the RIGHTS field can be believed. Clearly, an encryption function that mixes the bits thoroughly is required to ensure that tampering with the Rights Field also affects the known constant. EXCLUSIVE-OR'ing a constant with the concatenated RIGHTS and RANDOM fields will not do.

A second algorithm for protecting the RIGHTS field makes use of one-way functions, similar to the way ports are protected. When a server is asked to create a new object, it generates a random number, as usual. The RIGHTS field is then EXCLUSIVE-ORed with the random number and then used as the argument of the one-way function, F, yielding a value that is put into the RANDOM field of the capability. Symbolically,

RANDOM field = F(random-number XOR rights bits) .

The RIGHTS field is included in the capability itself in plaintext. When a capability arrives at the server, it finds the original random number from its internal tables and EXCLUSIVE-OR's the plaintext RIGHTS field with it, passing this result through F. If the result agrees with the RANDOM field in the capability, the capability is considered valid. Although a user can tamper with the plaintext RIGHTS field, such tampering will result in the server ultimately rejecting the capability.

When either of these protection systems are used, the owner of an object can easily give an exact copy of its capability to another process by just sending it the bit pattern, but to pass, say, read-only access, is slightly harder. To accomplish this task, the process must send the capability back to the server along with a bit mask and a request to fabricate a new capability with fewer rights.

This idea works well except that it requires going back to the server every time a sub-capability with fewer rights is needed. We will now describe a third algorithm that does not have this drawback. To start with, find a set of N commutative one-way functions, F_0, F_1, $\cdots$, F_{N-1} corresponding to the N rights present in the RIGHTS field. When an object is created, the server chooses a random number and puts it in both its internal table and the RANDOM field, just as in the very first scheme presented. The server also sets all

the RIGHTS field bits to 1.

A client can delete permission k from a capability by replacing the RANDOM field, R, with $F_k(R)$ and turning off the corresponding bit in the RIGHTS field. When a capability comes into the server to be used, the server fetches the original random number from its table, looks at the RIGHTS field and applies the functions corresponding to the deleted rights to it. If the result agrees with the number present in the capability, then the capability is accepted as genuine, otherwise it is rejected.

Note that although the RIGHTS field is not encrypted, it is pointless for a client to tamper with it, since the server will detect that. In theory at least, the RIGHTS field is not even needed, since the server could try all 2^N combinations of the functions to see if any worked. Its presence merely speeds up the checking. It should also be clear why the functions must be commutative—it does not matter in what order the bits in the RIGHTS field were turned off. This scheme is discussed in more detail in [MULLENDER 1985].

The organization of capabilities and objects discussed above has the interesting property that although no central record is kept of who has which capabilities, it is easy to revoke existing capabilities. All that the owner of an object need do is ask the server to change the random number stored in its internal table and return a new capability. Obviously this operation must be protected with a bit in the RIGHTS field, but if it succeeds, all existing capabilities for that object are instantly invalidated.

2.4. Protection without F-Boxes

Earlier we said that protection could also be achieved in software (i.e., without F-boxes). It is slightly more complicated, since it uses both conventional and public-key encryption [DIFFIE and HELLMAN 1976], but it is still quite usable. The basic idea underlying the method is the fact that in nearly all networks an intruder can forge nearly all parts of a message being sent except the source address, which is supplied by the network interface hardware. To take advantage of this property, imagine a (possibly symmetric) conceptual matrix, M, of conventional (e.g., DES) encryption keys, with the rows being labeled by source machine and the columns by destination machine. Thus the matrix selects a unique key for encrypting the *capabilities* in any message. The data need not be encrypted, although that is also possible if needed.

Each machine is assumed to know the contents of its row and column of the matrix, and nothing else (how this will be achieved will be discussed shortly). Thus a client C will know M_{CX} and M_{XC} for all X, and a server S will know M_{SX} and M_{XS}, all of which are conventional (not public) keys. With this arrangement, intruder I can easily capture messages from client C to server S, but attempts to "play them back" to the server will fail because the server will see the source machine as I (assumed unforgeable) and use element M_{IS} as the decryption key instead of the correct M_{CS}. No matter what the intruder does, he cannot trick the server into using a decryption key that decrypts the capabilities to make sense.

To avoid having to run the encryption/decryption algorithm frequently, all

machines can maintain a hashed cache of capabilities that they have been using frequently. Clients will hash their caches on the unencrypted capabilities in the form of triples: (unencrypted capability, destination, encrypted capabil it y), whereas servers will hash theirs in the form of triples: (encrypted capability, source, unencrypted capability).

To set up the matrix initially, the following procedure can be used. A public server, such as a file server, makes its put-port and a public encryption key known to the whole world. When a new machine joins the network (e.g., after a crash or upon initial system boot), it sends a broadcast message announcing its presence. Suppose, for example, the file server has just come up, and must (1) prove that it is the file server to other processes, and (2) establish the conventional keys used for encrypting capabilities in both directions.

A client machine, *C*, which receives the broadcast from the alleged file server, *F*, picks a new conventional encryption key, *K*, for use in subsequent *C* to *F* traffic and sends it to *F* encrypted with *F*'s public key. *F* then decrypts *K* and replies to *C* by sending a message containing both *K* and a newly chosen conventional key to be used for reverse traffic. This message is encrypted both with *K* itself and with the inverse of *F*'s public key, so *C* can use *K* and *F*'s public key to decrypt it. If the decrypted message contains *K*, *C* can be sure that the other conventional key was indeed generated by owner of *F*'s public key, thus convincing *C* that he is indeed talking to the file server. Both of the above-mentioned conditions have now been fulfilled, so normal communication can now take place. Note that the use of different conventional keys after each reboot make it impossible for an intruder to fool anyone by playing back old messages.

Yet another any possibility for protecting capabilities in the absence of F-box es is to use conventional link-level encryption on all the data communication lines.

3. Use of capabilities in Amoeba

In the preceeding sections we have seen how capabilities can be cryptographically protected so that they can be managed directly by user processes throughout the distributed system, without any help, or even knowledge by the operating system kernels. In the following sections we will look at some of the areas these capabilities have been applied in the Amoeba distributed operating system. The areas to be covered are: the memory server, the block server, the flat file server, the directory server, the multiversion file server, and the bank server. Capabilities are also used in other areas, but space limitations prevent them from being discussed here.

3.1. The memory server

The memory server is a process that manages physical memory and processes at the lowest level. It is actually part of the kernel present on each machine, but it communicates with other processes via the normal message protocol so that its clients do not perceive it as being special in any way.

The memory server is typically used for creating processes, as follows. The

parent process tells the memory server to CREATE SEGMENT, providing an initial size and some other information. The memory server then returns a capability for the newly created segment. Using this capability, the parent process can use the WRITE operation to load data into the segment (the READ operation can get it back again later if needed). The parent process will normally repeat this cycle, creating and loading segments until all the child process' initial segments have been constructed, for example, text, data, and stack segments.

To create the child process, the parent then performs a MAKE PROCESS operation, providing the capabilities for the child's segments as parameters. The memory server then returns a process capability for the child, with which the child can be started, stopped, and generally manipulated. By directing the CREATE SEGMENT requests to a memory server on a remote machine, the parent can create the child wherever it wants to, providing a more convenient and efficient interface than the traditional FORK + EXEC.

The memory server can also easily support an "electronic disk." An electronic disk of the required size is created using CREATE SEGMENT, and then can be read and written, either by local or remote processes using READ and WRITE.

3.2. The block server

The Amoeba file system also makes heavy use of capabilities. As far as the operating system is concerned, a file system is just one or more server processes, with no special privileges. This design makes it possible to have multiple, potentially quite different file systems running at the same time. Three distinct file systems have in fact been implemented.

The first file system is highly modular, consisting of a block server, flat file server, and directory server. The block server can be requested to allocate a disk block and return a capability for it. Using this capability, the block can be written, read, or deallocated. The block server has no concept of a file. By splitting the block server off from the file server, it becomes possible for any user to implement any kind of special-purpose file system that he needs, without having to get into the details of disk storage management.

3.3. The flat file server

The flat file server provides its clients with files consisting of a linear sequence of bytes, numbered from 0 to the file size - 1. The basic operations here are CREATE FILE, DESTROY FILE, WRITE FILE, and READ FILE. CREATE FILE returns a capability used in the other calls, each of which implicitly specifies a file via the capability, and a position in the file via a parameter. The server does not have any concept of an "open" file. One can operate on any file for which a valid capability can be presented.

3.4. The directory server

The directory server manages directories, each of which is a set of (ASCII name, capability) pairs. A typical operation is to present the directory server with the capability for a directory, plus an ASCII string, and ask it to look up and return the capability that corresponds to the given string in the given directory. Operations also exist to enter and remove (ASCII name, capability) entries from directories. These primitives, and a few others, provide an adequate basis for building up aribtrary directory trees, graphs, etc. Note that the capabilities within a directory need not all be file capabilities and certainly need not all be located in the same place or managed by the same server. To look up the path a/b/c relative to some directory, a client would ask the server to find the string "a" in that directory. If the capability returned happens to be for a directory managed by a different directory server, then the ensuing request to look up "b" just goes to the new server. Unless the client compared the SERVER fields in the two capabilities, it wouldn't even notice that succeeding requests were going to different servers. The distribution is completely transparent.

3.5. The multiversion file server

The second file system supports tree-shaped files. Each file consist of a tree of pages, rather than a simple linear byte sequence. An important property of this file system is its ability to provide atomic updates on files. In short, a user can ask to make a new version of a file, which results in a capability for the new version. The new version acts like it is a page-by-page copy of the original, although in fact, pages are only copied when they are changed.

The new version can be modified at will, and then atomically "committed," thus becoming the new file. A file is thus a sequence of versions. Once a version of a file has been committed, it cannot be modified. This technique has been designed for use with video disks and other "write once" media. More details can be found in [MULLENDER and TANENBAUM 1982].

The third file system is a capability-based UNIX file system, to ease the problem of moving existing applications from UNIX to Amoeba.

3.6. The bank server

Resource control and accounting also makes use of the capabilities. The basis for the resource control and accounting is the bank server, which manages "bank account" objects. The principal operation on bank accounts is transferring virtual money from one account to another. Thus to obtain permission to create a file, a client would present a capability for one of his accounts to the bank server, and request that the bank server withdraw some money from that account and deposit it in the account of the file server. Assuming the client trusts the file server, the client can pre-pay for a substantial amount of work, in order to eliminate the overhead of going back to the bank on each request.

The bank server is prepared to maintain accounts in different, possibly convertible, possibly inconvertible, currencies. This mechanism can form the basis of a variety of policies, used by different servers. For example, by having the

file server charge x dollars per kiloblock of disk space, quotas can be implemented by limiting how many dollars each client has. CPU time could be charged in francs, phototypesetter pages in yen, and so on. In some cases (e.g., disk blocks, but not typesetter pages), returning the resource might result in the client getting his money back

4. Discussion

In this paper we have shown how ports and capabilities can be managed in a protected way in a distributed operating system. By moving the entire capability management out of the kernel, we can provide a minimal kernel, and yet have a powerful and general conceptual basis for naming and protection throughout the system. A number of examples of how capabilities are used in Amoeba were presented as examples.

The Eden [Almes et al. 1985] and ACCENT [Rashid 1981] systems also use capability-like mechanisms for protection, butin both cases, the ultimately responsiblity for managing the capabilities rests with the kernel. In Eden, users may manage capabilities directly, but the kernel maintains copies, to be able to verify each one before it is used. We maintain that moving all of the capability management out of the kernel is a step in the right direction. Just as file servers are now rarely part of the kernel of distributed systems, capability management should not be either. The smaller and simpler the kernel, the easier it is to write, debug, and maintain. Furthermore, if the system consists of a building full of rooms with wall sockets into which any user can plug any machine, protection based on trusted kernels managing capabilities becomes impossible. A malicious user could modify his kernel to subvert the capability checking and thereby bypass the protection scheme.

In [Donnelley 1981], a description is given of work being done at Lawrence Livermore Laboratory is given. Two schemes are described, one using a password in each capability, and one using public key cryptography. Although these schemes are similar to ours in some ways, they do not provide a way to protect individual rights bits to allow one capability to read an object and another to write it. Furthermore, our proposal addresses the problem of how to prevent users from impersonating servers or reading network traffic not intended for them. Both the F-boxes and the matrix method described in 2.4 can be used to fight wiretapping.

References

Almes, G.T, Black, A.P., Lazowska, E.D. and Noe, J.D. "The Eden System,: A Technical Review," *IEEE Trans Softw. Eng.*, vol. SE-11, pp. 43-59, Jan. 1985.

Birrell, A.D. and Nelson, B.J. "Implementing Remote Procedure Calls," *ACM Trans. Comp. Syst.*, vol. 2, pp. 39-59, Feb. 1984.

Dennis, J.B. and Van Horn, E.C. "Programming Semantics for Multiprogrammed Computations," *Comm. ACM*, vol. 9, pp. 143-154, March 1966.

DIFFIE, W. and HELLMAN, M.E. "New Directions in Cryptography," *IEEE Trans. Inf. Theory*, vol. IT-22, pp. 644-654, Nov. 1976.

DONNELLEY, J.E. "Managing Domains in a Network Operating System," *Proc. Conf on Local Networks and Distributed Office Systems*, Online, pp. 345-361, 1981.

EVANS, A., KANTROWITZ, W. and WEISS, E. "A User Authentication Scheme not Requiring Security in the Computer," *Commun. ACM*, vol. 17, pp. 437-442, Aug. 1974.

LEVY, H. *Capability-Based Computer Systems,* Bedford, Mass.: Digital Press, 1984.

MULLENDER, S.J. "Principles of Distributed Operating System Design" Ph.D. thesis, Vrije Universiteit, Amsterdam, 1985.

MULLENDER, S.J. and TANENBAUM, A.S. "Protection and Resource Control in Distributed Operating Systems," *Computer Networks* (to appear in 1985).

MULLENDER, S.J. and TANENBAUM, A.S. "A Distributed File Server Based on Optimistic Concurrency Control," Report IR-80, Wiskundig Seminarium, Vrije Universiteit, 32 pp. Nov. 1982.

MULLENDER, S.J. and VITANYI, P.M.B. "Distributed Match-Making for Processes in Computer Networks," Report CS-8424, Centrum v Wiskunde en Informatica, Dec. 1984.

NELSON, B.J. "Remote Procedure Call," Tech. Rep. CSL-81-9, Xerox PARC, 1981.

PURDY, G.B. "A High-Security Log-in Procedure," *Commun. ACM*, vol. 17, pp. 442-445, Aug. 1974.

RASHID, R.F. and ROBERTSON, G.G. "Accent: A Communication Oriented Network Operating System Kernel," *Proc. Eigth Symp. on Operating Syst. Prin.*, ACM, pp. 64-75, 1981.

SPECTOR, A.Z. "Performing Remote Operations Efficiently on a Local Computer Network," *Comm. ACM.*, vol. 25, pp. 246-260, April 1982.

TANENBAUM, A.S. and MULLENDER, S.J. "The Design of a Capability-Based Distributed Operating System," *Computer Journal*, (to appear in 1985).

WILKES, M.V. *Time Sharing Computer Systems*, New York: American Elsevier, 1968.

Capability-Based Protection in Distributed Operating Systems

Andrew S. Tanenbaum
Department of Mathematics and Computer Science
Amsterdam, The Netherlands

Robbert van Renesse
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Sape J. Mullender
Centre for Mathematics and Computer Science
Amsterdam, The Netherlands

Capability-based operating systems have traditionally required large, complex kernels to manage the use of capabilities. In our proposal, capability management is done entirely by user programs without giving up any of the protection aspects normally associated with capabilities. The basic idea is to use one-way functions and encryption to protect sensitive information.

1. Introduction

Soon, most office buildings will have a cable snaking through the cable ducts, with an outlet in each room into which users can plug their personal computers. The traditional approach to protection, a secure operating system in every machine to check permissions before carrying out a command, is not suitable for such an environment. It is too easy for a malicious user to replace the operating system in one of the network machines, or to replace a machine altogether by one without a secure operating system, to obtain confidential information illicitly.

New methods for protection must be devised, methods that do not require secure, trustworthy operating systems. This paper presents mechanisms, based on encryption. We shall show that they are equally powerful, and, in some cases, more versatile than existing protection schemes, implemented by a

Capability-Based Protection in Distributed Operating Systems
A. S. Tanenbaum, R. van Renesse, and S. J. Mullender
Proceedings of Symposium Certificering van Software
Utrecht, Netherlands
November 1984

secure operating system. We propose to base the software design on a different conceptual model - the object model. In this model, the system deals with abstract objects, each of which has some set of abstract operations that can be performed on it.

Associated with each object are one or more "capabilities" [Dennis66] which are used to control access to the object, both in terms of who may use the object and what operations he may perform on it. At the user level, the basic system primitive is performing an operation on an object, rather than such things as establishing connections, sending and receiving messages, and closing connections. For example, a typical object is the file, with operations to read and write portions of it.

The object model is well-known in the programming languages community under the name of "abstract data type." This model is especially well-suited to a distributed system because in many cases an abstract data type can be implemented on one of the processor-memory modules described above. When a user process executes one of the visible functions in an abstract data type, the system arranges for the necessary underlying message transport from the user's machine to that of the abstract data type and back. The header of the message can specify which operation is to be performed on which object. This arrangement gives a very clear separation between users and objects, and makes it impossible for a user to directly inspect the representation of an abstract data type by bypassing the functional interface.

A major advantage of the object or abstract data type model is that the semantics are inherently location independent. The concept of performing an operation on an object does not require the user to be aware of where objects are located or how the communication is actually implemented. This property gives the system the possibility of moving objects around to position them close to where they are frequently used. Furthermore, the issue of how many processes are involved in carrying out an operation, and where they are located is also hidden from the user.

It is frequently convenient to implement the object model in terms of clients (users) who send messages to services. A service is defined by a set of commands and responses. Each service is handled by one or more server processes that accept messages from clients, carry out the required work, and send back replies. The design of these servers and the design of the protocols they use form an important part of the system software of our proposed fifth generation computers.

As an example of the problems that must be solved, consider a file server. Among other design issues that must be dealt with are how and where information is stored, how and when it is moved, how it is backed up, how concurrent reads and writes are controlled, how local caches are maintained, how information is named, and how accounting and protection are accomplished. Furthermore, the internal structure of the service must be designed: how many server processes are there, where are they located, how and when do they communicate, what happens when one of them fails, how is a server process organized internally for both reliability and high performance, and so on.

Analogous questions arise for all the other servers that comprise the basic system software.

2. Ports and capabilities

2.1. Ports

Every service has one or more *ports* [Mullender84] to which client processes can send packets to contact the service. Ports consist of large numbers, typically 48 bits, which are known only to the server processes that comprise the service, and to the service's clients. For a public service, such as the system file service, the port will be generally made known to all users. The ports used by an ordinary user process will, in general, be kept secret. Knowledge of a port is taken by the system as prima facie evidence that the sender has a right to communicate with the service. Of course the service is not required to carry out work for clients just because they know the port, for example, the public file service may refuse to read or write files for clients lacking account numbers, appropriate authorization, etc.

Although the port mechanism provides a convenient way to provide partial authentication of clients ("if you know the port, you may at least talk to the service"), it does not deal with the authentication of servers. The basic primitive operations offered by the system are PUT(PORT, MESSAGE) and GET(PORT, MESSAGE). Since everyone knows the port of the file server, as an example, how does one insure that malicious users do not execute GETs on the file server's port, in effect impersonating the file server to the rest of the system?

One approach is to have all ports manipulated by kernels that are presumed trustworthy and are supposed to know who may GET from which port. We reject this strategy because some machines, *e.g.*, personal computers connected to larger multimodule systems may not be trustworthy, and also because we believe that by making the kernel as small as possible, we can enhance the reliability of the system as a whole. Instead, we have chosen a different solution that can be implemented in either hardware or software. First we will describe the hardware solution; later we will describe the software solution.

In the hardware solution, we need to place a small interface box, which we call an F-box (Function-box) between each processor module and the network. The most logical place to put it is on the VLSI chip that is used to interface to the network. Alternatively, it can be put on a small printed circuit board inside the wall socket through which personal computers attach to the network. In those cases where the processors have user mode and kernel mode and a trusted operating system running in kernel mode, it can also be put into operating system software. In any event, we assume that somehow or other all packets entering and leaving every processor undergo a simple transformation that users cannot bypass.

The transformation works like this. Each port is really a pair of ports, P, and G, related by: $P = F(G)$, where F is a (publicly-known) one-way function [Wilkes68, Purdy74, Evans74] performed by the F-box. The one-way function

has the property that given G it is a straightforward computation to find P, but that given P, finding G is so difficult that the only approach is to try every possible G to see which one produces P. If P and G contain sufficient bits, this approach can be made to take millions of years on the world's largest supercomputer, thus making it effectively impossible to find G given only P. Note that a one-way function differs from a cryptographic transformation in the sense that the latter must have an inverse to be useful, but the former has been carefully chosen so that no inverse can be found.

Using the one-way F-box, the server authentication can be handled in a simple way. Each server chooses a get-port, G, and computes the corresponding put-port, P. The get-port is kept secret; the put-port is distributed to potential clients, or, in the case of public servers, is published. When the server is ready to accept client requests, it does a GET(G). The F-box then computes $P = F(G)$ and waits for packets containing P to arrive. When one arrives, it is given to the process that did GET(G). To send a packet to the server, the client merely does PUT(P), which sends a packet containing P in a header field to the server. The F-box on the sender's side does not perform any transformation on the P field of the outgoing packet.

Now let us consider the system from an intruder's point of view. To impersonate a server, the intruder must do GET(G). However, G is a well-kept secret, and is never transmitted on the network, Since we have assumed that G cannot be deduced from P (the one-way property of F) and that the intruder cannot circumvent the F-box, he cannot intercept packets not intended for him. Replies from the server to the client are protected the same way, only with the client picking a get-port for the reply, say, G', and including $P' = F(G')$ in the request packet.

The presence of the F-box makes it easy to implement digital signatures for still further authentication, if that is desired. To do so, each client chooses a random signature, S, and publishes $F(S)$. The F-box must be designed to work as follows. Each packet presented to the F-box contains three special header fields: destination (P), reply (G'), and signature (S). The F-box applies the one-way function to the second and third of these, transmitting the three ports as: P, $F(G')$, and $F(S)$, respectively. The first is used by the receiver's F-box to admit only packets for which the corresponding GET has been done, the second is used as the put-port for the reply, and the third can be used to authenticate the sender, since only the true owner of the signature will know what number to put in the third field to insure that the publicly-known $F(S)$ comes out.

It is important to note that the F-box arrangement merely provides a simple mechanism for implementing security and protection, but gives operating system designers considerable latitude for choosing various policies. The mechanism is sufficiently flexible and general that it should be possible to put it into hardware with precluding many as-yet-unthought-of operating systems to be designed in the future.

2.2. Capabilities

In any object-based system, a mechanism is needed to keep track of which processes may access which objects and in what way. The normal way is to associate a capability with each object, with bits in the capability indicating which operations the holder of the capability may perform. In a distributed system this mechanism should itself be distributed, that is, not centralized in a single monolithic "capability manager." In our proposed scheme, each object is managed by some service, which is a user (as opposed to kernel) program, and which understands the capabilities for its objects.

A capability typically consists of four fields:

1. The put-port of the service that manages the object
2. An Object Number meaningful only to the service managing the object
3. A Rights Field, which contains a 1 bit for each permitted operation
4. A Random Number for protecting each object

The basic model of how capabilities are used can be illustrated by a simple example: a client wishes to create a file using the file service, write some data into the file, and then give another client permission to read (but not modify) the file just written. To start with, the client sends a packet to the file service's put-port specifying that a file is to be created. The request might contain a file name, account number and similar attributes, depending on the exact nature of the file service. The server would then pick a random number, store this number in its object table, and insert it into the newly-formed object capability. The reply would contain this capability for the newly created (empty) file.

To write the file, the client would send a succession of data packets, each one containing the capability and some data. When each WRITE request arrived at the file server process, the server would normally use the object number contained in the capability as as index into its tables to locate the file.

Several object protection systems are possible using this framework. In the simplest one, the server merely compares the random number in the file table (put there by the server when the object was created) to the one contained in the capability. If they agree, the capability is assumed to be genuine, and all operations on the file are allowed. This system is easy to implement, but does not distinguish between READ, WRITE, DELETE, and other operations that may be performed on objects.

However, it can easily be modified to provide that distinction. In the modified version, when a file (object) is created, the random number chosen and stored in the file table is used as an encryption/decryption key. The capability is built up by taking the Rights Field (*e.g.*, 8 bits), which is initially all 1s indicating that all operations are legal, and the Random Number Field (*e.g.*, 56 bits), which contains a known constant, say, 0, and treating them as a single number. This number is then encrypted by the key just stored in the file table, and the result put into the newly minted capability in the combined Rights-Random Field. When the capability is returned for use, the server uses the object number (not encrypted) to find the file table and hence the encryption/decryption key. If the result of decrypting the capability leads to

the known constant in the Random Number Field, the capability is almost assuredly valid, and the Rights Field can be believed. Clearly, an encryption function that mixes the bits thoroughly is required to ensure that tampering with the Rights Field also affects the known constant. Exclusive or'ing a constant with the concatenated Rights and Random fields will not do.

When this modified protection system is used, the owner of the object can easily give an exact copy of the capability to another process by just sending it the bit pattern, but to pass, say, read-only access, is harder. To accomplish this task, the process must send the capability back to the server along with a bit mask and a request to fabricate a new capability whose Rights Field is the Boolean-and of the Rights Field in the capability and the bit mask. By choosing the bit mask carefully, the capability owner can mask out any operations that the recipient is not permitted to carry out.

This modified system works well except that it requires going back to the server every time a sub-capability with fewer rights is needed. We have devised yet another protection system that does not have this drawback. This third scheme requires the use of a set of N commutative one-way functions, $F_0, F_1, \cdots, F_{N-1}$, corresponding to the N rights present in the Rights Field. When an object is created, the server chooses a random number and puts it in both the file table and the Random Number Field, just as in the first scheme presented. It also sets all the Rights Field bits to 1.

A client can delete permission k from a capability by replacing the random number, R, with $F_k(R)$ and turning off the corresponding bit in the Rights Field. When a capability comes into the server to be used, the server fetches the original random number from the file table, looks at the Rights Field, and applies the functions corresponding to the deleted rights to it. If the result agrees with the number present in the capability, then the capability is accepted as genuine, otherwise it is rejected. Note that although the Rights Field is not encrypted, it is pointless for a client to tamper with it, since the server will detect than immediately. In theory at least, the Rights Field is not even needed, since the server could try all 2^N combinations of the functions to see if any worked. Its presence merely speeds up the checking. It should also be clear why the functions must be commutativeit does not matter in what order the bits in the Rights Field were turned off.

The organization of capabilities and objects discussed above has the interesting property that although no central record is kept of who has which capabilities, it is easy to retract existing capabilities. All that the owner of an object need do is ask the server to change the random number stored in the file table. Obviously this operation must be protected with a bit in the Rights Field, but if it succeeds, all existing capabilities are instantly invalidated.

2.3. Protection without F-boxes

Earlier we said that protection could also be achieved without F-boxes. It is slightly more complicated, since it uses both conventional and public-key encryption, but it is still quite usable. The basic idea underlying the method is the fact that in nearly all networks an intruder can forge nearly all parts of a packet being sent except the source address, which is supplied by the network interface hardware. To take advantage of this property, imagine a (possibly symmetric) conceptual matrix of conventional (*e.g.*, DES) encryption keys, with the rows being labeled by source machine and the columns by destination machine. Thus the matrix selects a unique key for encrypting the capabilities in any packet. The data need not be encrypted, although that is also possible if needed.

Each machine is assumed to know its row and column of the matrix, and nothing else (how this will be achieved will be discussed shortly). With this arrangement, intruder I can easily capture packets from client C to server S, but attempts to "play them back" to the server will fail because the server will see the source machine as I (assumed unforgeable) and use element M_{IS} as the decryption key instead of the correct M_{CS}. No matter what the intruder does, he cannot trick the server into using a decryption key that decrypts the capabilities to make sense, that is, to contain random numbers that agree with those stored in the file tables.

To avoid having to run the encryption/decryption algorithm frequently, all machines can maintain a hashed cache of capabilities that they have been using frequently. Clients will hash their caches on the unencrypted capabilities in the form of triples: (unencrypted capability, destination, encrypted capability), whereas servers will hash theirs in the form of triples: (encrypted capability, source, unencrypted capability).

To set up the matrix initially, the following procedure can be used. A public server, such as a file server, makes its put-port and a public encryption key known to the whole world. When a new machine joins the network (*e.g.*, after a crash or upon initial system boot), it sends a broadcast message announcing its presence. Suppose, for example, the file server has just come up, and must (1) prove that it is the file server to other processes, and (2) establish the conventional keys used for encrypting capabilities in both directions.

A client machine, C, which receives the broadcast from the alleged file server, F, picks a new conventional encryption key, K, for use in subsequent C to F traffic and sends it to F encrypted with F's public key. F then decrypts K and replies to C by sending a packet containing both K and a newly chosen conventional key to be used for reverse traffic. This packet is encrypted both with K itself and with the inverse of F's public key, so C can use K and F's public key to decrypt it. If the decrypted packet contains K, C can be sure that the other conventional key was indeed generated by owner of F's public key, thus convincing C that he is indeed talking to the file server. Both of the above-mentioned conditions have now been fulfilled, so normal communication can now take place. Note that the use of different conventional keys after each reboot make it impossible for an intruder to fool anyone by playing back old packets.

SUMMARY

This paper has discussed a model for a fifth generation computer system architecture and its operating system. The operating system is based on the use of objects protected by sparse capabilities. Conclusions

The paper shows that it is possible and practical to build capability-based distributed operating systems, with capability management outside of the operating system kernel. Since the operating system itself is particularly vulnerable to attack in an office environment as we have described, our method is more secure than traditional protection schemes that must rely on the security of the operating system kernel.

Two methods have been presented for the implementation of authenticated communication between client and server processes, one using F-boxes, the other using a combination of public key encryption and conventional encryption techniques. Currently public key encryption is still expensive, both in terms of computational effort and storage requirements. The F-box mechanism is a good alternative until fast public key algorithms arrive. F-boxes can be put in the cable ducts, on the network interface cards, in integrated circuits that carry out the network protocol, or, if necessary, in the operating system kernel.

Capability management need not be carried out by a secure operating system: all operations on capabilities that are currently implemented in secure operating system kernels can also be carried out by choosing appropriate encryption techniques, with which client processes can be allowed to handle capabilities and carry out certain (restricted) sets of operations on them.

REFERENCES

[Dennis66]
DENNIS, J. B. and HORN, E. C. VAN, "Programming Semantics for Multiprogrammed Computation," *Comm. ACM*, vol. 9, no. 3, pp.143-155, March 1966.

[Evans74]
EVANS, A., KANTROWITZ, W., and WEISS, E., "A User Authentication Scheme Not Requiring Secrecy in the Computer," *Comm. ACM*, vol. 17, no. 8, pp.437-442, August 1974.

[Mullender84]
MULLENDER, S. J. and TANENBAUM, A. S., "Protection and Resource Control in Distributed Operating Systems," *Computer Networks*, vol. 8, no. 5,6, pp.421-432, 1984.

[Purdy74]
PURDY, G. B., "A High Security Log-in Procedure," *Comm. ACM*, vol. 17, no. 8, pp.442-445, August 1974.

[Wilkes68]
WILKES, M. V., *Time-Sharing Computer Systems*. New York: American Elsevier, 1968.

Protocols

A Secure High-Speed Transaction Protocol

Sape J. Mullender
Centre for Mathematics and Computer Science
Amsterdam, The Netherlands

Robbert van Renesse
Vrije Universiteit
Amsterdam, The Netherlands

Most computer networks use a byte stream protocol for communication between processes, which suffer from two important drawbacks: the addressing mechanisms provided are often process-dependent or location-dependent, and communication is slow. While carrying out research into distributed operating systems at the Vrije Universiteit and the Centre for Mathematics & Computer Science, we have developed a transaction-oriented transport protocol for the ***Amoeba*** distributed operating system [Mullender86], aimed for high-speed, with an addressing mechanism that is not only more general, but provides a protection mechanism as well. The basic mechanism for communication between processes is the transaction: a client process sends a request to a server process, which carries out the request and returns a reply. Protection is provided by using ports, chosen from a sparse address space, for addressing services. These ports serve as a "capability" for communication with the service. Through its simplicity, the transaction protocol achieves much higher transmission rates than other protocols executing on similar hardware (about 300 Kbytes/sec process-to-process).

The protection mechanism will be described, and the mechanisms for realising high transmission speeds.

1980 Mathematics Subject Classification: 68A05, 68B20.
1982 CR Categories: C.2.2, C.2.4, D.4.4.
Keywords & Phrases: transaction protocols, connectionless protocols, capabilities, local-area networks.

1. Introduction

Traditional networks and distributed systems are based on the concept of two processes or processors communicating via connections. The connections are typically managed by a hierarchy of complex protocols, usually leading to complex software and extreme inefficiency. (An effective transfer rate of 0.1 megabit/sec over a 10 megabit/sec local network, which is only 1% utilization,

is frequently barely achievable.)

We reject this traditional approach of viewing a distributed system as a collection of discrete processes communicating via multilayer (e.g., ISO) protocols, not only because it is inefficient, but because it puts too much emphasis on specific processes, and by inference, on processors. Instead we propose to base the software design on a different conceptual modelthe object model. In this model, the system deals with abstract objects, each of which has some set of abstract operations that can be performed on it.

Associated with each object are one or more "capabilities" [Dennis66] which are used to control access to the object, both in terms of who may use the object and what operations he may perform on it. At the user level, the basic system primitive is performing an operation on an object, rather than such things as establishing connections, sending and receiving messages, and closing connections. For example, a typical object is the file, with operations to read and write portions of it.

The object model is well-known in the programming languages community under the name of "abstract data type." This model is especially well-suited to a distributed system because in many cases an abstract data type can be implemented on one of the processor-memory modules described above. When a user process executes one of the visible functions in an abstract data type, the system arranges for the necessary underlying message transport from the user's machine to that of the abstract data type and back. The header of the message can specify which operation is to be performed on which object. This arrangement gives a very clear separation between users and objects, and makes it impossible for a user to inspect the representation of an abstract data type directly by bypassing the functional interface.

A major advantage of the object or abstract data type model is that the semantics are inherently location independent. The concept of performing an operation on an object does not require the user to be aware of where objects are located or how the communication is actually implemented. This property gives the system the possibility of moving objects around to position them close to where they are frequently used. Furthermore, the issue of how many processes are involved in carrying out an operation, and where they are located is also hidden from the user.

It is frequently convenient to *implement* the object model in terms of clients (users) who send messages to services. A service is defined by a set of commands and responses. Each service is handled by one or more server processes that accept messages from clients, carry out the required work, and send back replies. The design of these servers and the design of the protocols they use form an important part of the system software of our proposed fifth generation computers.

As an example of the problems that must be solved, consider a file server. Among other design issues that must be dealt with are how and where information is stored, how and when it is moved, how it is backed up, how concurrent reads and writes are controlled, how local caches are maintained, how information is named, and how accounting and protection are accomplished.

Furthermore, the internal structure of the service must be designed: how many server processes are there, where are they located, how and when do they communicate, what happens when one of them fails, how is a server process organized internally for both reliability and high performance, and so on. Analogous questions arise for all the other servers that comprise the basic system software.

2. PROTECTION

Every service has one or more *ports* [Mullender84] to which client processes can send messages to contact the service. Ports consist of large numbers, typically 48 bits, which are known only to the server processes that comprise the service, and to the service's clients. For a public service, such as the system file service, the port will be generally made known to all users. The ports used by an ordinary user process will, in general, be kept secret. Knowledge of a port is taken by the system as prima facie evidence that the sender has a right to communicate with the service. Of course the service is not required to carry out work for clients just because they know the port, for example, the public file service may refuse to read or write files for clients lacking account numbers, appropriate authorization, etc.

Although the port mechanism provides a convenient way to provide partial authentication of clients ("if you know the port, you may at least talk to the service"), it does not deal with the authentication of servers. The basic primitive operations offered by the system are put(port, message) and get(port, message). Since everyone knows the port of the file server, as an example, how does one insure that malicious users do not execute gets on the file server's port, in effect impersonating the file server to the rest of the system?

One approach is to have all ports manipulated by kernels that are presumed trustworthy and are supposed to know who may get from which port. We reject this strategy because some machines, e.g., personal computers connected to larger multimodule systems may not be trustworthy, and also because we believe that by making the kernel as small as possible, we can enhance the reliability of the system as a whole. Instead, we have chosen a different solution that can be implemented in either hardware orif necessaryin software.

In the hardware solution, we need to place a small interface box, which we call an F-box (Function-box) between each processor module and the network. The most logical place to put it is on the VLSI chip that is used to interface to the network. Alternatively, it can be put on a small printed circuit board inside the wall socket through which personal computers attach to the network. In those cases where the processors have user mode and kernel mode and a trusted operating system running in kernel mode, it can also be put into operating system software. In any event, we assume that somehow or other all packets entering and leaving every processor undergo a simple transformation that users cannot bypass.

The transformation works like this. Each port is really a pair of ports, P, and G, related by: $P = F(G)$, where F is a (publicly-known) one-way function [Wilkes68, Purdy74, Evans74] performed by the F-box. The one-way function

has the property that given G it is a straightforward computation to find P, but that given P, finding G is so difficult that the only approach is to try every possible G to see which one produces P. If P and G contain sufficient bits, this approach can be made to take millions of years on the world's largest supercomputer, thus making it effectively impossible to find G given only P. Note that a one-way function differs from a cryptographic transformation in the sense that the latter must have an inverse to be useful, but the former has been carefully chosen so that no inverse can be found.

Using the one-way F-box, the server authentication can be handled in a simple way. Each server chooses a get-port, G, and computes the corresponding put-port, P. The get-port is kept secret; the put-port is distributed to potential clients or in the case of public servers, is published. When the server is ready to accept client requests, it does a get(G). The F-box then computes $P = F(G)$ and waits for packets containing P to arrive. When one arrives, it is given to the process that did get(G). To send a packet to the server, the client merely does put(P), which sends a packet containing P in a header field to the server. The F-box on the sender's side does not perform any transformation on the P field of the outgoing packet.

Now let us consider the system from an intruder's point of view. To impersonate a server, the intruder must do get(G). However, G is a well-kept secret, and is never transmitted on the network, Since we have assumed that G cannot be deduced from P (the one-way property of F) and that the intruder cannot circumvent the F-box, he cannot intercept packets not intended for him. Replies from the server to the client are protected the same way, only with the client picking a get-port for the reply, say, G', and including $P' = F(G')$ in the request packet.

The presence of the F-box makes it easy to implement digital signatures for still further authentication, if that is desired. To do so, each client chooses a random signature, S, and publishes $F(S)$. The F-box must be designed to work as follows. Each packet presented to the F-box contains three special header fields: destination (P), reply (G'), and signature (S). The F-box applies the one-way function to the second and third of these, transmitting the three ports as: P, $F(G')$, and $F(S)$, respectively. The first is used by the receiver's F-box to admit only packets for which the corresponding get has been done, the second is used as the put-port for the reply, and the third can be used to authenticate the sender, since only the true owner of the signature will know what number to put in the third field to insure that the publicly-known $F(S)$ comes out.

It is important to note that the F-box arrangement merely provides a simple *mechanism* for implementing security and protection, but gives operating system designers considerable latitude for choosing various *policies*. The mechanism is sufficiently flexible and general that it should be possible to put it into hardware with precluding many as-yet-unthought-of operating systems to be designed in the future.

3. COMMUNICATION PRIMITIVES

In the literature about computer networks, one finds much discussion of the ISO OSI reference model [Zimmermann80] these days. It is our belief that the price that must be paid in terms of complexity and performance in order to achieve an "open" system in the ISO sense is much too high, so we have developed a much simpler set of communication primitives, which we will now describe.

3.1. Transaction vs. stream communication

Most distributed systems have a connection mechanism that is based on the idea of two processes going to some effort to set up a connection, using the connection, and then tearing it down. The assumption is that a connection will be used for a stream of information so long that the overhead needed to set it up and tear it down are basically negligible. Most streams will consist of a file of one kind or anothera source program, a binary program, an input file, and so on. To see how long the average file is, we have conducted some measurements on the UNIX† system used in our department by the faculty and staff for research (no students, thus). The results of these measurements show that 34% of all files are less than 512 bytes, 52% are less than 1K bytes, 67% are less than 2K bytes, 79% are less than 4K bytes, 88% are less than 8K bytes, and 94% are less than 16K bytes.

The above considerations have led us to a different approach [Mullender83]. With packets of even 2K bytes, two thirds of all files fit into a single packet. Consequently, it is much simpler to adopt a "Request-Reply" or "Transaction" style of communication, in which the basic primitive is the client sending a request to a server and the server sending a reply back to the client. The client uses trans and the server getreq and putrep. Trans sends a request, and blocks until a reply is received. Getreq blocks the server until a request is received, which can then be processed, after which a reply can be sent using putrep. Each request-reply pair is completely self-contained, and independent of any other ones that may previously been sent. In other words, no concept of a "connection" exists. Not only is this conceptually much more appropriate for use in an operating system, but it is much simpler to implement than a complex 7-layer protocol, not to mention offering lower delay. Henceforth we will refer to a request-reply pair as a *transaction*, which is not to be confused with transactions with a data base.

3.2. Basic communication protocol

Instead of a 7-layer protocol, we effectively have a 4-layer protocol. The bottom layer is the Physical Layer, and deals with the electrical, mechanical and similar aspects of the network hardware. The next layer is the Port Layer, and deals with the location of ports, the transport of (32K byte) datagrams (packets whose delivery is not guaranteed) from source to destination and enforces

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

the protection mechanism of the previous section. On top of this we have a layer that deals with the reliable transport of bounded length (32K byte) requests and replies between client and server. We have called this layer the Transaction Layer. The final layer has to do with the semantics of the requests and replies, for example, given that one can talk to the file server, what commands does it understand.

Since systems of the kind we are describing will use high-speed, highly reliable local networks, few if any of the complex mechanisms designed for flow- and error-control in long-haul networks are useful here. Among other things, a simple stop-and-wait protocol is sufficient. The main function of the Transaction Layer is to provide an end-to-end *message* service built on top of the underlying *datagram* service, the main difference being that the former uses timers and acknowledgements to guarantee delivery whereas the latter does not.

The Transaction Layer protocol is straightforward. When the client does a trans, a packet containing the request is sent to the server and a timer is started. If the server does not acknowledge receipt of the request packet before the timer expires (usually by sending the reply, but in some special cases by sending a separate acknowledgement packet), the Transaction Layer retransmits the packet again and restarts the timer. When the reply finally comes in, the client sends back an acknowledgement (usually piggybacked onto the next request packet) to allow the server to release any resources, such as buffers, that were acquired for this transaction. Under normal circumstances, reading a long file, for example, consists of the sequence

From client : request for block 0
From server: here is block 0
From client : acknowledgement for block 0 and request for block 1
From server: here is block 1
etc.

The protocol can handle the situation of a server crashing and being rebooted quite easily since each request contains the capability for the file to be read and the position in the file to start reading. Between requests, the server has no "activation record" or other table entry whose loss during a crash causes the server to forget which files were open, etc., because no concept of an open file or a current position in a file exists on the server's side. Each new request is completely self-contained. Of course for efficiency reasons, a server may keep a cache of frequently accessed i-nodes, file blocks etc., but these are not essential and their loss during a crash will merely slow the server down slightly while they are being dynamically refreshed after a reboot.

4. The Port Layer

The Port Layer is responsible for the speedy transmission of 32K byte datagrams. The Port Layer need only do this reasonably reliably, and does not have to make an effort to guarantee the correct delivery of every datagram. This is the responsibility of the Transaction Layer. Our results show that this approach leads to significantly higher transmission speeds, due to simpler protocols.

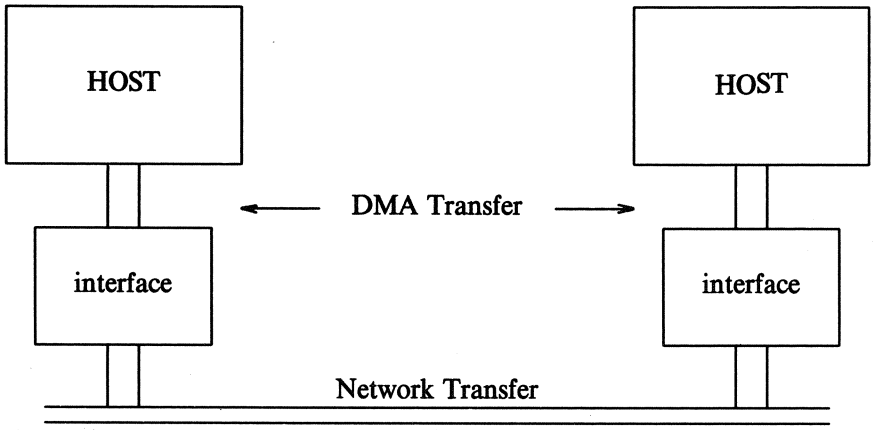


FIGURE 1. A typical local-area network interface.

Theoretically, very high speeds are achievable in modern local-area networks. A typical example of a local-area network interface is shown in Fig. 1. When a host transmits a packet to another host, the packet is first transferred to the interface by means of a *direct memory access* (DMA) transfer. When this is done, the packet is transmitted over the network. After the packet has been received by the destination interface, it can be transferred to the destination host's memory, again using a DMA transfer. While this transfer is going on, the sending host may already transfer the next packet to the interface. A sequence of packets is thus transmitted by interchanging periods of DMA transfers and network transfers. On most interfaces DMA transfers and network transfers cannot occur simultaneously.

It is now simple to deduce an upper bound for the maximum transfer rate over the network: A typical speed for DMA transfers is 1 byte/μsec, and the typical transmission speed of a 10 Mbit local-area network is also 1 byte/μsec. Since DMA transfer and network transfer cannot overlap, but DMA at the destination host *can* overlap with the DMA of the next packet at the source host, an upper bound for the transfer rate of a typical local-area network is 500,000 bytes/sec point-to-point.

Obviously, to achieve such a transmission rate, the overhead of the protocol must be kept as low as possible, while an effort must be made to overlap DMA s at both communicating parties. To achieve this, we have chosen a very large datagram size for the Port Layer, which has to split up the

datagrams into small packets that the network hardware can cope with. This approach allows the implementor of the Port Layer to exploit the possibilities that the hardware has to offer to achieve an efficient stream of packets.

Our Port Layer interfaces to a 10 Mbit token ring that allows *scatter-gather*; that is, a packet can be sent to or from the interface in several DMA transfers, and then transmitted over the network separately. We discovered that this allows us to do two important things to speed up the protocol. First, when a packet is received, the header can be inspected separately, so the protocol can decide where in memory the packet must go. The protocol can then transfer the packet directly from the interface to the right place in memory, without having to copy it. A copy loop would halve the transmission speed. Second, the separation of DMA and transmission allows the protocol to prepare a transmission by doing the DMA . The transmission can then be initiated immediately when the signal is received that the receiver is ready. In our implementation of the Port Layer these considerations have resulted in the protocol that will now be described.

The transmitter begins by transferring and sending the first 2K of the datagram to be transmitted (2K is the maximum packet size allowed by the hardware). Immediately after the transmission is complete, the DMA for the next 2K bytes is started, but it is not yet transmitted. In the mean time, the receiver is interrupted by the arrival of the first packet. It extracts the header, examines it and decides where the body of the packet should go. Then the body of the packet is transferred from the interface to its final location in memory. While this is being done, the receiver prepares a tiny *acknowledgement* packet to tell the transmitter it is prepared for the next packet. As soon as the DMA transfer of the previous packet has finished, this acknowledgement is sent back to the transmitter. When the transmitter receives it, the transfer of the next packet to the interface will have finished, so it can then be sent immediately. This sequence is continued until the whole datagram is transmitted.

5. The transaction layer

It is the responsibility of the Transaction Layer to guarantee the arrival of requests and replies. The Transaction Layer makes use of the Port Layer and timers to achieve this.

The interface to the transaction layer basically consists of three calls, one for clients, and two for servers. All calls use a small datastructure, called **Mref**, which contains a pointer to a small fixed-size out-of-band buffer for the transmission of commands and parameters to the server, a pointer to the main body of data to be transferred, and the length of the main body of data (0 to 32768), as follows:

```
typedef struct Mref {
        char    *M_oob;
        char    *M_buf;
        unsigned M_len;
} Mref;
```

The client, in order to do a transaction calls

```
trans(cap, req, rep);
        Cap *cap; Mref *req, *rep;
```

The server receives requests and sends replies with

```
getreq(port, cap, req);
        Port *port; Cap *cap; Mref *req;

putrep(rep);
        Mref *rep;
```

In principle, the Transaction Layer works as follows: When a client calls **trans**, the Transaction Layer generates a *reply-port* to enable the server to send a reply. The server port is deduced from the capability; the first 48 bits of the capability for an object identify the service that controls the object. The request is then sent, using **put**, and a *retransmission timer* is started.

The server, which previously had made a call to **getreq**, receives the request; the capability is filled in, and the received message is put in the buffers referred to by **req**. As soon as the request is received, the server's Transaction Layer starts a *piggyback timer*. When the server has not sent a reply before this timer expires, a separate acknowledgement is sent to put the client at ease, and stop its retransmission timer. When the server sends a reply to the client the same thing happens, more or less, with the role of client and server reversed. When a client makes a sequence of transactions with a single server, a subsequent request will acknowledge receipt of the previous reply.

The client maintains one more timer, the *crash timer*. This timer is set when the server's acknowledgement to a request has been received, and is used to detect server crashes. Whenever this timer expires, the client sends an "are you still alive?" packet to the server, to which the server replies with an acknowledgement.

When transactions occur quickly, one after the other, no extra acknowledgements are sent at all. Only when transactions take a long time (say, longer than a minute), acknowledgements are sent, and when transactions take much longer than that (say, ten minutes) then "are you still alive" messages begin to be sent.

5.1. Timer management

If the timers are started and stopped in exactly the way described above, the Transaction Layer would become unacceptably slow. Per (quick) transaction, two retransmission timers and two piggyback timers would have to be started and stopped, eight timer actions altogether.

There is a much more efficient way of dealing with timers, one that makes use of a *sweep algorithm*. This algorithm does not implement very accurate timers, but accuracy of the timer intervals is not very important to the correct and efficient operation of the protocol.

The sweep algorithm is called every n clock tics. N must be chosen such that n tics is about the minimum timer interval needed (the piggyback timer interval). Whenever the algorithm is called, it makes a sweep over all outstanding transactions. If the state of a transaction has changed, the new state is recorded. If it has not changed, a counter is incremented, telling for how long the state has remained the same. If the (state, counter) combination has reached a certain value, the sweep algorithm carries out the appropriate actions, usually sending an acknowledgement, retransmitting a message, or aborting a transaction.

Because this algorithm is used there is no code needed in the transaction code itself, reducing the overhead of the Transaction Layer significantly. In this way, the code executed in the Transaction Layer is optimised for the normal case (no errors).

5.2. Results

Two versions of the algorithm have now been implemented. The one described has been implemented on the ***Amoeba*** distributed operating system, and achieves over 300,000 bytes a second from user process to user process (using M68000s and a Pro-net ring). It is now being implemented under UNIX where we expect to obtain more than 200,000 bytes/sec, assuming the communicating processes are not swapped.

An older version of the protocol, using 2K byte datagrams, now gets 90,000 bytes/sec across the network between two VAX-750s running a normal load of work, without causing a significant load on the system itself.

Several services, implemented under UNIX, are using the Transaction Layer interface, and it is our experience that these services are easy to design and that they work efficiently.

The *port* mechanism allows us to move services from one machine to another, completely transparently to the user. The F-boxes do not yet exist in hardware, but are built into the operating system. The one-way function does not significantly slow the system down, because a cache is maintained of get-

port/put-port pairs.

REFERENCES

[Dennis66]
DENNIS, J. B. and HORN, E. C. VAN, "Programming Semantics for Multiprogrammed Computation," *Comm. ACM*, vol. 9, no. 3, pp.143-155, March 1966.

[Evans74]
EVANS, A., KANTROWITZ, W., and WEISS, E., "A User Authentication Scheme Not Requiring Secrecy in the Computer," *Comm. ACM*, vol. 17, no. 8, pp.437-442, August 1974.

[Mullender83]
MULLENDER, SAPE J., RENESSE, ROBBERT VAN, and TANENBAUM, ANDREW S., "A Transaction-Oriented Transport Protocol", internal paper, Centre for Mathematics and Computer Science, 1983.

[Mullender84]
MULLENDER, S. J. and TANENBAUM, A. S., "Protection and Resource Control in Distributed Operating Systems," *Computer Networks*, vol. 8, no. 5,6, pp.421-432, 1984.

[Mullender86]
MULLENDER, S. J. and TANENBAUM, A. S., "The Design of a Capability-Based Distributed Operating System," *The Computer Journal*, vol. 29, no. 4, pp.289-300, 1986.

[Purdy74]
PURDY, G. B., "A High Security Log-in Procedure," *Comm. ACM*, vol. 17, no. 8, pp.442-445, August 1974.

[Wilkes68]
WILKES, M. V., *Time-Sharing Computer Systems*. New York: American Elsevier, 1968.

[Zimmermann80]
ZIMMERMANN, H., "OSI Reference Model—The ISO Model of Architecture for Open Systems Intercon nection," *IEEE Trans. Comm.*, vol. COM-28, pp.425-432, April 1980.

Distributed Match-Making for Processes in Computer Networks

Sape J. Mullender
Paul M.B. Vitányi
Centre for Mathematics and Computer Science
Amsterdam, The Netherlands

In the very large multiprocessor systems and, on a grander scale, computer networks now emerging, processes are not tied to fixed processors but run on processors taken from a pool of processors. Processors are released when a process dies, migrates or when the process crashes. In distributed operating systems using the service concept, processes can be clients asking for a service, servers giving a service or both. Establishing communication between a process asking for a service and a process giving that service, without centralized control in a distributed environment with mobile processes, constitutes the problem of distributed match-making. Logically, such a match-making phase precedes routing in store-and-forward computer networks of this type. Algorithms for distributed match-making are developed and their complexity is investigated in terms of message passes and in terms of storage needed. The theoretical limitations of distributed match-making are established, and the techniques are applied to several network topologies.

1. Introduction

We investigate the problem of setting up communication-when-needed between processes in a multiprocessor network where processes have *names* but no permanent *addresses*. A mechanism for this purpose is called a *name-server*, analogous to the telephone system's directory assistance server: given a *name* it returns an *address*. A single *centralized* name server in the network can be taken out through a single processor crash, thereby effectively killing all communication and crashing the entire network. A more robust solution is *distributing* the name server. A great variety of options and problems of both theoretical and practical interest are attached to this issue. Our motivation was provided by the design objectives of the *Amoeba* distributed operating system project [Mullender86].

Distributed Match-Making for Processes in Computer Networks
S. J. Mullender and P. M. B. Vitanyi
Proceedings 4th ACM Principles of Distributed Computing
Minaki, Canada
August 1985

1.1. The catering service problem

Suppose you want to give a party in your Silicon Valley home, but do not care for the bother. You want a catering service. Now it so happens, that you do not know the address or telephone number of such a service. Anyway, even if you did, this would not do you much good. In Silicon Valley such small outfits come and go so fast that it is unlikely that this service, which you used two years ago, still exists at the old address. You can phone them, but the number gets you somebody who has never heard of your old catering service. There are several courses of action you can take.

- One way to solve your problem is to send mail to everybody in town asking whether they supply catering service. In computer networks this is called *broadcasting*.
- Another way is to wait until you get an advertisement leaflet of a catering service in your mailbox. Below we call this *sweeping*.

Most likely, you do one of the following:

- You look in the Yellow Pages under the appropriate heading. If everybody exclusively uses YP for all services then we may view the YP outfit as a centralized name server. Services reveal their whereabouts by advertising there and clients look them up there. If the YP company crashes then clients and services cannot be matched anymore, and society grinds to a halt.
- You buy a suitable newspaper and look up "catering" in the advertisement section. Now the name server is distributed. Catering services advertise in many newspapers. If one newspaper flounders, this will not create problems for you.
- You ask some of your friends whether *they* know where to find the desired service. Some of your friends crashing will not prevent you finding a caterer. The name server is distributed in this case as well, and, depending on how sociable you are, perhaps better.

Having found the address or telephone number of a catering service, you have to find a way to route your request to them. Thus, match-making between clients and services necessarily precedes routing in a mobile society. Note that the catering service, in order to execute the task you set them, may call on other services such as a car rental service. The catering service then is a client with respect to the car rental service. Clearly, everybody can be server, client or both.

1.2. Multiprocessors and computer networks

New generation computers must be fast, reliable, and flexible. One way to achieve this is to build them from a small number of basic processor-memory modules that can be assembled together to realize machines of various sizes. The use of multiple modules can make the machines not only fast, but also achieve a substantial amount of fault tolerance. The primary difference between machines should be the number of modules, rather than the type of

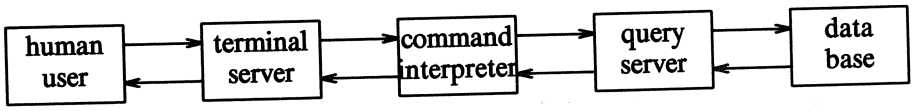
the modules. In principle, any of these machines can be gracefully increased in size to improve performance by adding new modules or decreased in size to allow removal and repair of defective modules. The software running on the various machines should be in essence identical. It should be possible to connect different machines together to form even larger machines and to partition existing machines into disjoint pieces when necessary, all in a way transparent to the user level software. When a user has a heavy computation to do, an appropriate number of processor-memory modules are temporarily assigned to him. When the computation is completed, they are returned to the idle pool for use by other users. Note that in this view a *computer network* is essentially such machine on a grand scale.

Software design for these new machines can advantageously be based on the *object model.* In this model, the system deals with abstract *objects*, each of which has some set of abstract *operations* that can be performed on it. At the user level, the basic system primitive is performing an operation on an object, rather than such things as establishing connections, sending and receiving messages, and closing connections. For example, a typical object is the file, with operations to read and write portions of it. The object model is also known under the name of "abstract data type" [Liskov74]. A major advantage of the object or abstract data type model is that the semantics are inherently location independent. The concept of performing an operation on an object does not require the user to be aware of where objects are located or how the communication is actually implemented. This property gives the system the possibility of moving objects around to position them close to where they are frequently used. Furthermore, the issue of how many processes are involved in carrying out an operation, and where they are located is also hidden from the user.

1.3. The service model

It is convenient to *implement* the object model in terms of clients (users) who send messages to services [Mullender86]. A service is defined by a set of commands and responses. Each service is handled by one or more server processes that accept messages from clients, carry out the required work, and send back replies.

> As an example, consider a *file server.* The design must deal with how and where information is stored, how and when it is moved, how it is backed up, how concurrent reads and writes are controlled, how local caches are maintained, how information is named, and how accounting and protection are accomplished. The internal structure of the service must be designed: how many server processes are there, where are they located, how and when do they communicate, what happens when one of them fails, how is a server process organized internally for both reliability and high performance, and so on. A server can itself be client to another service. The possible hierarchy of services is the strength of the model:



A crash of the database server, will be detected by the query server, which must then try to recover from it. The query server can retry the request, it might rephrase a query to get the answer from another database server, and as a last resort, it can report failure to its client, the command interpreter. In this way the human client at the top of the hierarchy gets to cope only with irrecoverable errors and crashes in the system.

More precisely, *Services* are offered by a number of *server* processes, distributed over the network. *Client* processes send *requests* to services; the services carry out these requests and return a *reply*. Essentially, every job in the system is executed by a dynamic network of servers executing each other's requests. So a process can be a client, a server, or both, and change its role dynamically. New services can be created by installing server processes for them. Services can be removed by destroying their server processes (or by making them stop behaving like a server, i.e., by telling them to stop receiving requests). Server processes can be migrated through the network, either by actually moving the process from one host to another, or only in effect, by destroying the server process in one host and creating another one in a different host at the same time. A specific service may be offered by one, or by more than one server process. In the latter case, we assume that all server processes that belong to one service are equivalent: a client sees the same result, regardless which server process carries out its request. A process resides in a network *node*. Each node has an *address* and we assume that, given an address, the network is capable of routing a message to the node at that address. A service is identified by its *port*. A port uniquely names a service. We shall therefore also refer to a service by its port. Ports give no clue about the physical location of a server process.

1.4. The problem of Match-Making

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. We can view match-making as yet another service in the system, be it the *primus inter pares*. Thus, we need to implement a *name server* to serve a connection between client process and server process.

A *centralized* 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 host of the name server crashes, the entire network crashes. This solution also causes an overload of messages in the neighborhood of the host.

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

centralized one. 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. 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 [Dalal77]).

We investigate realizations of name servers in the entire range between centralized 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 $O(\sqrt{n})$ message passes, by using limited numbers of point-to-point messages.

1.5. 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*.

1.6. Outline of the paper.

We develop a class of distributed algorithms for match-making between client processes and server processes in computer networks. We investigate the expected performance of such an algorithm under random choices. Subsequently, we determine the optimal lower bound on the performance in number of message passes or "hops" for any such algorithm, in any network, under any strategy, distributed or not. This yields a combinatorial lemma which may be interesting in its own right, and 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. We consider criteria for robustness. Second, we apply the method to particular networks, both designed networks and spontaneously emerged networks. Finally, a probabilistic and a hashing algorithm for match-making are investigated.

1.7. Related work.

Distributed match-making between *clients* and *servers* will be used in the *Amoeba* distributed operating system [Mullender86]. Essentially the Manhattan topology method below has been used before in the torus-shaped Stony Brook Microcomputer Network [Gelernter82]. Some current multiprocessor systems avoid the communication overload due to mobile processes, which use

broadcasting to do the match-making, by opting for the processes to run on fixed processors [Seitz85]. Other system designers have chosen for mobile processes, but use the crash-vulnerable solution of a centralized name server [Needham82]. The present paper introduces, and systematically explores for the first time, the general concept of distributed match-making.

2. A THEORY OF DISTRIBUTED MATCH-MAKING

Below we obtain lower bounds on the message pass complexity of a class of Locate algorithms (called Shotgun Locate), for the entire range from centralized 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 many network topologies.

2.1. Framework for shotgun locate

The networks we consider 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 neighbors, performs local computations on messages and sends messages to neighbors. All these actions take finite time. A *message pass* or *hop* consists of the sending of a message from one node to one of its direct neighbors.

1. 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.
2. For each network $G=(U,E)$ and associated match-making algorithm, there are total functions P, Q such that:

$$P, Q: U \rightarrow 2^U .$$

   (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.
3. 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). We can timestamp the messages 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.

2.2. Probabilistic analysis

Let the number of elements in a given set U (universe) of nodes 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 $p+q \geq 2\sqrt{n}$. 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 a successful match-making for *each* pair.

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

- The method deterministically yields success.
- We get by with $p+q < 2\sqrt{n}$.

2.3. 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 $r_{i,j}$ ($1 \leq i,j \leq n$) is the *rendez-vous* matrix. Each entry $r_{i,j}$, in the ith row and jth column of R, 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:

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

To prevent waste in message passes, we can take care that the inclusions in (M1) are replaced by equalities. (But then the surviving subnetwork after a node crash may lack this property again.) 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 $r_{i,j}$, *cf.* § 2.4.)

2.3.1. Examples of rendez-vous matrices associated with both well-known and lesser known strategies.

1. *Broadcasting.* The server stays put and client looks everywhere:

		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. *Centralized 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. The nodes are hierarchically ordered by 1,2,3<7; 4,5,6<8; 7,8<9:

		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

2.3.2. Lower bound

There are n possible *rendez-vous* nodes and n^2 elements in R. By choice of P, 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 number of message passes for a match-making instance can be as low as $\log n$. 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,

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

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

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

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:

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

We now proceed to derive an exact lower bound on $m(n)$ expressed in terms of the number k_i of times node i occurs in R, i.e., is used as *rendez-vous* for a pair of nodes ($1 \leqslant i \leqslant n$).

PROPOSITION 1.
Consider the rendez-vous matrix R as defined. Then the average value $\frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \#P(i)\#Q(j)$ is bounded below by:

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

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 and else $\rho_{i,j} = 0$, and let $\gamma_{i,j} = 1$ if node i occurs in column j and else $\gamma_{i,j} = 0$, ($1 \leqslant i,j \leqslant n$). 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

$$R_i C_i \geqslant k_i \ . \tag{3}$$

Furthermore, since

$$\begin{aligned} k_j R_i^2 - 2\sqrt{k_i k_j}\, R_i R_j + k_i R_j^2 &= (\sqrt{k_j} R_i - \sqrt{k_i} R_j)^2 \\ &\geqslant 0 \ , \end{aligned}$$

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

$$\frac{k_j R_i}{R_j} + \frac{k_i R_j}{R_i} \geqslant 2\sqrt{k_i k_j} \ ,$$

from which it follows that:

$$\sum_{i=1}^{n} R_i \sum_{j=1}^{n} k_j R_j^{-1} \geqslant \sum_{i=1}^{n} \sum_{j=1}^{n} \sqrt{k_i k_j} \ . \tag{4}$$

Hence,

$$\begin{aligned} \sum_{i=1}^{n} \sum_{j=1}^{n} \#P(i) \#Q(j) &\geqslant \sum_{i=1}^{n} \sum_{j=1}^{n} r_i c_j \quad \text{(by (M1) \& (1))} \\ &= \sum_{i=1}^{n} r_i \times \sum_{j=1}^{n} c_j \\ &= \sum_{i=1}^{n} R_i \times \sum_{j=1}^{n} C_j \quad \text{(by (2))} \\ &\geqslant \sum_{i=1}^{n} R_i \sum_{j=1}^{n} k_j R_j^{-1} \quad \text{(by (3))} \\ &\geqslant \left[\sum_{i=1}^{n} \sqrt{k_i} \right]^2 \quad \text{(by (4)),} \end{aligned}$$

which yields the Proposition. □

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

> We can adjust the distributed match-making strategy to the relative frequency of these happenings, so as to minimize 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 minimize $m(n)$ replacing (M3) by (M3′):
>
> $$m(i,j) = \#P(i) + \alpha_{i,j} \#Q(j) \ . \tag{M3′}$$

Proposition 1 immediately gives us 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,

$$\sum_{i=1}^{n} \sum_{j=1}^{n} (r_i + c_j) = n \sum_{i=1}^{n} (r_i + c_i)$$
$$< 2n \sum_{i=1}^{n} \sqrt{k_i} \ .$$

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. □

It is not difficult to see that 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.

2.3.3. Truly distributed match-making, centralized link-server
Propositions 1 and 2 specialize to the corollary below for $k_1 = k_2 = \cdots = k_n = n$, the *truly distributed case.* Here, each node occurs equally often as *rendez-vous* node in matrix R, and hence carries an equal load of the 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 lower bound we saw before in the probabilistic approach. 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 centralized name*

server, we obtain:

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

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

2.3.4. Upper bound for complete networks

For complete networks the above lower bounds on the number of message passes for match-making are about sharp. For instance:

PROPOSITION 3.

For the truly distributed case arrangements can be constructed such that the lower bounds are (nearly) matched by upper bounds. Viz., for each complete network there exists functions P, Q such that, for all $1 \leqslant i,j \leqslant n$, $\#P(i)\#Q(j) \approx n$, $\#P(i)+\#Q(j) \approx 2\sqrt{n}$, *and* $k_i \approx n$.

PROOFSKETCH.

Arrange the *rendez-vous* matrix R as a checker board consisting of (as near as possible) $\sqrt{n} \times \sqrt{n}$ squares, or nearly squares, of about n entries each. Each square is filled with about n copies of one unique node out of the n nodes, a different one for each square; *cf.* Example 4. □

PROPOSITION 4.

Let R be the rendez-vous matrix for an n-node network. Let k_i $(1 \leqslant i \leqslant n)$ *be the multiplicity of node i in R, and let* $m(n)$ *be the average match-making cost associated with R. We can lift this strategy to a 4n-node network by constructing a* $4n \times 4n$ *rendez-vous matrix* R' *with* $k_i' = 4k_{i \bmod n}$ *the multiplicity of node i in* R' $(1 \leqslant i \leqslant 4n)$ *and* $m'(4n) = 2m(n)$ *the associated average match-making cost.*

PROOF.

Replace each entry $r_{i,j}$ of R by a 2×2 submatrix consisting of 4 copies of $r_{i,j}$. The resulting $2n \times 2n$ matrix is M. Let R_i $(i = 1,2,3,4)$ be four, pairwise element disjoint, isomorphic copies of M. Consider the $4n \times 4n$ matrix R':

$$R' = \begin{bmatrix} R_1 & R_2 \\ R_3 & R_4 \end{bmatrix} .$$

The number of distinct nodes in R' is 16 times that in R and $k_i' = 4k_{i \bmod n}$ $(1 \leqslant i \leqslant 4n)$. It is easy to see that the $(2i \bmod 2n)$th column [row] of R' contains twice as many distinct nodes as the $(i \bmod n)$th column [row] of R $(1 \leqslant i \leqslant 2n)$. Therefore, the average match-making cost associated with R' is $m'(4n) = 2m(n)$. □

The most *inefficient* match-making strategy is $P(i) = Q(j) = U$ $(1 \leqslant i,j \leqslant n)$, yielding $m(n) = 2n$.

2.3.5. Upper bound for non-Complete networks

The topology of a network $G=(U,E)$ determines the overhead in message passes needed for routing a message to its destination. For the complete networks we have considered, the number of message passes $m(i,j)$ for a match-making between a service at node i and a client at node j equals $\#P(i)+\#Q(j)$. If the subgraph induced by the sets $P(i)$, $Q(j)$ $(1\leqslant i,j\leqslant n)$ is connected, and $i\in P(i)$ and $j\in Q(j)$, and we broadcast the messages over spanning trees in these subgraphs, then the number of message passes $m(i,j)$ equals the number of addressed nodes $\#P(i)+\#Q(j)$. Otherwise, there is an overhead $m(i,j)-\#P(i)-\#Q(j)>0$ of message passes for routing messages from i,j to $P(i)$, $Q(j)$. In designing distributed name servers for non-complete networks, the achievable message pass efficiency of match-making very much depends on how far we can reduce this overhead. For this reason, in a *ring network*, *no* match-making algorithms can do significantly better than broadcasting (i.e., $m(n)\in\Omega(n)$).

2.4. Robustness, fault-tolerance and efficiency

In computer networks, and also in multiprocessor systems, the communication algorithms must be able to cope with faulty processors, crashed processors, broken communication links, reconfigured network topology and similar issues. A centralized name server (Example 3) is very *efficient*, but if its host crashes the whole network fails. It is one of the advantages of truly distributed algorithms that they may continue in the presence of faults. With respect to implementing the name server, we can distinguish two distinct criteria for robustness.

- The name server should be *distributed* in the sense that no number of node crashes, which leaves a surviving network, can prevent surviving clients from locating surviving servers offering a desired service (for instance, by first moving to another address). This rules out a centralized name server, but the distributed Examples 1, 2, 4, 5, 6 are fine. It is lack of robustness according to this criterion that makes the efficient Hash Locate (last section) so fragile.
- The name server should be *redundant* in the sense that no number of node crashes can prevent a client at a surviving node from locating a service offered at a surviving node. For example, the Shotgun algorithm expounded above, may be locally incapacitated by a *rendez-vous* node crashing. We can remedy this situation by choosing P and Q such that, for all $1\leqslant i,j\leqslant n$,

$$\#(P(i)\cap Q(j))\geqslant f+1\ ,$$

where f is the maximal number of faults at any time in the network. (There remains of course the problem of how, or whether it is still possible, to route the match-making messages to their destinations in the surviving subnetwork.) The safest solution is obviously $P(i)\cap Q(j)=U$ $(1\leqslant i,j\leqslant n)$. This criterion holds equally for Shotgun Locate and Hash

Locate.

Robustness is *inefficient* and has a price tag in number of message passes per match-making instance. That question is not addressed in this paper.

3. Implementations in particular networks

We assume that each node has a table containing the names of all other nodes together with the minimum cost to reach them and the neighbor at which the minimum cost path starts. In [Erdös70] a construction is given to divide every connected graph in $O(\sqrt{n})$ disjoint *connected* subgraphs of $\leqslant \sqrt{n}$ nodes each. Number the nodes in each subgraph 1 through $\sqrt{n}$ (if necessary, divide the excess numbers over the nodes). Each node i has a table containing the route to the next (adjacent) node i. In the worst case such a path consists of $2\sqrt{n}$ message passes. (Each of the connected subgraphs contains at most $\sqrt{n}$ nodes. The shortest path, between the two nodes labelled i in two adjacent connected subgraphs, is therefore not longer than $2\sqrt{n}$.)

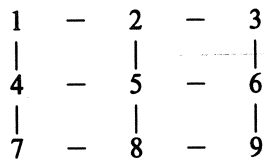
Server's Algorithm. A server at the node labelled i in one of the subgraphs communicates its (port, address) to all nodes i in the remaining $O(\sqrt{n})$ subgraphs . It follows from above that this takes $O(n)$ message passes. Size $O(\sqrt{n})$ suffices for the cache of each node.

Client's Algorithm. A client broadcasts for a service (along a spanning tree) in the subgraph where it resides. This takes at most $\sqrt{n}$ message passes.

Under the practical assumption that clients need to locate services usually far more frequently than servers need to post (port, address), this scheme is fairly optimal. Additionally, the caches are kept to a moderate size. Moreover, in practice, many store-and-forward networks will require but $O(\sqrt{n})$ message passes on the average to broadcast over the required subsets of $\sqrt{n}$ nodes of the server's algorithm. All this suggests that in most networks using this method the average number of message passes per match-making instance can be substantially less than the order n figure. In the remainder of this section we look at match-making in some networks with specific topologies.

3.1. Manhattan networks

The network is laid out as a $p \times q$ rectangular grid of nodes. Post availability of a service along its row and request a service along the column the client is on. Caches are of size $O(q)$ and 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 [Gelernter82]. In the obvious generalization to d-dimensional meshes the method takes $m(n)=2n^{(d-1)/d}$ message passes.

3.2. Multidimensional cubes

The network $G=(U,E)$ is a d-dimensional cube with U the set of nodes of the cube with addresses of d bits and E the set of edges which connect nodes of which the addresses differ in a single bit. $n=\#U=2^d$ and $\#E=d2^{d-1}$. Assume that d is even.

Server's Algorithm. A server at an address $s=s_1s_2\cdots s_d$ broadcasts its (port, address) along a spanning tree to all nodes in the $d/2$-dimensional cube spanned by the nodes in

$$P(s) = \{a_1a_2\ldots a_{\frac{d}{2}}s_{\frac{d}{2}+1}\ldots s_d \mid a_1,\ldots,a_{\frac{d}{2}} \in \{0,1\}\} \ .$$

Client's Algorithm. A client at an address $c=c_1c_2\cdots c_d$ broadcasts its query along a spanning tree to all nodes in the $d/2$-dimensional cube spanned by the nodes in

$$Q(c) = \{c_1c_2...c_{\frac{d}{2}}a_{\frac{d}{2}+1}...a_d \mid a_{\frac{d}{2}+1},...,a_d \in \{0,1\}\} \ .$$

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

$$P(s) \bigcap Q(c) = \{c_1c_2...c_{\frac{d}{2}}s_{\frac{d}{2}+1}...s_d\} \ .$$

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 $\sqrt{n}$-size caches. Variants of the algorithm are obtained by splitting the corner address used in the algorithm not in the middle but in pieces of ϵd and $(1-\epsilon)d$ bits. *Cf.* Example 6. For instance, to adapt the method to take advantage of relative immobility of servers, to get lower average. Excessive clogging at intermediate nodes may be prevented by sending messages to a random address first, to be forwarded to their true destination second [Valiant82].

3.3. Fast permutation networks

For various reasons [Broomell83] fast permutation networks like the *Cube-Connected Cycles* network are important interconnection patterns. An algorithm similar to that of the d-dimensional cube yields, appropriately tuned, for an n-node CCC network caches of size $\sqrt{n/\log n}$ and $m(n) \in O(\sqrt{n \log n})$.

3.4. 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. 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 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.

3.5. 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 $k=\frac{1}{2}\log n$. This message pass complexity is much better than $\Omega(\sqrt{n})$, but the cache size towards the top of the hierarchy increases rapidly. Essentially, the cache of a node may need 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.

In the *Amoeba* distributed operating system, for instance, even the operating system itself is accessed just like any other service [Mullender86]. "Operating System Service" is thus a local service, useful only to local clients. Clients on other hosts must use similar services, local to their host.

The *Amoeba* system provides a way for services to restrict the availability of the service they offer to some local group of processes, the processes within the host where the service resides, the processes within the local-area network of the service, within the campus network, etc. This last model seems the most likely model for the interaction between clients and services. Nearly every service will be a local service in some sense, with only few services being truly global. Under these assumptions, the burden of the processing of locate postings and requests can be distributed more or less evenly over the hosts at each level of the network hierarchy. This is essentially the generalization presented later in the section on Hash Locate.

3.6. Existing networks

Many wide-area computer networks are not completely designed at the outset but grow and change dynamically. Yet one can identify common characteristics.

- The network resembles an undirected tree with a core in which we can imagine the root, and with some additional edges thrown in. It appears that UUCPnet (the anarchistic network connecting most UNIX† systems) has this form in the sense that the number of extra edges thrown in are not more than the the number of nodes in a spanning tree. The extra edges would typically occur between geographically near nodes.
- The degree of the nodes should not be too large. Ideally bounded by a constant. Yet nodes nearer to the core of the tree tend to be of higher degree. Compare *backbone* sites, *feeder* sites and *terminal* sites in UUCPnet. The hierarchy of the nodes towards the core is very pronounced as can be seen in the table. The degree of super-backbone sites like *ihnp4* is over 600, of backbone sites like *decvax* 40 and *mcvax* 45, and a feeder site like *sdcsvax* is 17. Terminal sites like *ace* have degree 1.
- The network is planar to a large extent. This reflects the geographical cost factor but also the tree aspect mentioned above. Thus, the ARPAnet, to a large extent predesigned, is approximately planar and even the chaotic UUCPnet is not too unplanar.

In the table below we have collected some statistics about the state of the known sites of UUCPnet at August 15, 1984. The total number of sites of UUCPnet is 1916 and of EUnet (European part) 153. The total number of edges in UUCPnet is 3848 and in EUnet 211. The degree of the nodes varies between the unlikely number 0 (one such node is appropriately named *loyalist*) and 641 (which is *ihnp4*, in real life AT&T in Naperville). In the table below we list the number of nodes having a given degree.

Let us consider trees as described above. The number of nodes in the

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

#sites	degree	#sites	degree
25	0	3	25
840	1	1	27
384	2	2	28
207	3	2	30
115	4	2	32
83	5	1	33
71	6	2	34
32	7	1	35
29	8	2	36
11	9	1	37
17	10	1	38
5	11	1	39
7	12	1	40
14	13	1	42
10	14	1	43
6	15	1	44
2	16	3	45
2	17	1	46
3	18	1	47
3	19	1	52
3	20	2	63
3	21	1	70
4	22	1	471
3	23	1	641
3	24		

balanced tree is n, the number of levels is l with the root at level l and the leaves at level 0, and the degree of nodes at the i-th level is $d(i)$. Then a 'factorial' relation holds:

$$d(l)d(l-1)\cdots d(1) = n \ .$$

Setting $d(l) = cl^{1+\epsilon}$, for constants $c, \epsilon > 0$, yields $c^l(l!)^{1+\epsilon} = n$. By Stirling's approximation, we get after some calculation:

$$l \sim \frac{\log n}{(1+\epsilon)\log\log n} \ .$$

If the exponent $1+\epsilon$ in the expression for $d(m)$ is doubled then the depth of the tree is halved for the same number of nodes.
Setting $d(l) = c2^{el}$, for constants $c, \epsilon > 0$ yields:

$$n = c^l 2^{\sum_{i=1}^{l} \epsilon i} = c^l 2^{\frac{\epsilon}{2} l^2} \ .$$

Therefore,

$$l = \frac{\sqrt{\log^2 c + 2\epsilon \log n} - \log c}{\epsilon}$$

(The logarithms have base 2.) If ϵ is quadrupled then the depth of the tree is halved for the same number of nodes.

The strategy in such trees can be simple: all services advertise at the path leading to the root of the tree, and similarly the clients request services on the path to the root of the tree. Then the average number of message passes used for each match-making instance, is $m(n) \in O(l)$. The cache at each node needs to be of the order of the number of elements in the subtree of which it is the root. For smaller caches the older and less used entries can be discarded in favour of new ones, leading to a Lighthouse Locate like algorithm (see below). It may seem that such large caches are unrealistic and that, anyway, in distributed networks all nodes should be symmetric. However, even in a genuinely distributed and anarchistically growing network as UUCPnet a hierarchy of nodes develops according to the node degree (number of links with other nodes in the network). This points to the fact that nodes higher in the hierarchy must dedicate more computing power and memory to running the network. Hence it is not unrealistic to have the cache size increase for nodes higher in the hierarchy.

4. Lighthouse locate

We imagine the processors as discrete coordinate points in the 2-dimensional Euclidean plane grid spanned by $(\epsilon,0)$ and $(0,\epsilon)$. The number of servers satisfying a particular port in an n-element region of the grid has expected value sn for some fixed constant $s>0$.

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. 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 govern the length of the locate beam (and its duration) by the sequence

$$1213121412131215121312141213121612131 2 \cdots$$

Here the length of the locate beam is il once in each interval of 2^i trials. (This sequence is sequence 51 in Sloane's catalogue [Sloane73].) 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 il. This schedule has the additional profit that the servers which

drift nearer to the client are located with less time-loss. Note that in a sequence of 2^k trials there are 2^{k-i} length *il* trials ($1 \leqslant i \leqslant k$).

> 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 [Dalal78] 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 neighbor. This neighbor, upon reception of such a message decreases the hop count (in the message) by 1, 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.

5. Hash locate and beyond

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 like in the Shotgun Locate but using the port identities as well:

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

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 neighborhoods. A neighborhood 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 specialization of this more general method.

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

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

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 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. Moreover, if all *rendez-vous* nodes for a particular service crash then this *takes out completely* that particular service from the entire network. If the service is indispensable, the entire network crashes. In this sense Hash Locate is far more vulnerable to node crashes than the more distributed versions of Shotgun Locate. Examples 1, 2 and 3 may also be viewed as borderline examples of Hash Locate. Examples 4, 5 and 6 are not Hash Locate methods, since Hash Locate cannot be distributed in this genuine sense.

Two obvious approaches can make Hash Locate more robust for node crashes. First, the hash function can map a service name onto many different network addresses for added reliability. Second, when the *rendez-vous* node for a particular service is down, rehashing can come up with another network address to act as a backup *rendez-vous* node. It then becomes necessary that services regularly poll their *rendez-vous* nodes to see if they are still alive.

References

[Broomell83]
BROOMELL, G. and HEATH, J. R., "Classification Categories and Historical Development of Circuit Switching Topologies," *ACM Computing Surveys*, vol. 15, no. 2, pp.95-133, June 1983.

[Dalal77]
DALAL, Y. K., "Broadcast Protocols in Packet Switched Computer Networks", Ph.D. Dissertation, Computer Science Dept., Stanford University, Stanford, Calif., April 1977.

[Dalal78]
DALAL, Y.K. and METCALFE, R.M., "Reverse Path Forwarding of Broadcast Packets," *Comm. ACM*, vol. 21, no. 12, pp.1040-1048, December 1978.

[Erdös70]
ERDÖS, P., GERÉNCSER, L., and MATÉ, A., "Problems of Graph Theory Concerning Optimal Design," pp. 317-325 in Colloquium Math. Soc. Janos Bolyai 4: Combinatorial Theory and its Applications, ed. V. T. Sós, North-Holland Publishing Company, Amsterdam (1970).

[Gelernter82]
GELERNTER, D. and BERNSTEIN, A. J., "Distributed Communication via a Global Buffer," *Proc. 1st ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp.10-18, 1982.

[Liskov74]
LISKOV, B. and ZILLES, S., "Programming with Abstract Data Types," *SIGPLAN Notices*, vol. 9, pp.50-59, April 1974.

[Mullender86]

MULLENDER, S. J. and TANENBAUM, A. S., "The Design of a Capability-Based Distributed Operating System," *The Computer Journal*, vol. 29, no. 4, pp.289-300, 1986.

[Needham82]
NEEDHAM, R. M. and HERBERT, A. J., *The Cambridge Distributed Computer System*. Reading, Ma.: Addison-Wesley, 1982.

[Seitz85]
SEITZ, CH. L., "The Cosmic Cube," *Comm. ACM*, vol. 28, pp.22-33, 1985.

[Sloane73]
SLOANE, N. J. A., *A Handbook of Integer Sequences*. New York: Academic Press, 1973.

[Valiant82]
VALIANT, L.G., "A Scheme for Fast Parallel Communication," *SIAM Journal on Computing*, vol. 11, no. 2, pp.350-361, May 1982.

Reliability

Reliability Issues in Distributed Operating Systems*

Andrew S. Tanenbaum
Robbert van Renesse
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Distributed systems span a wide spectrum in the design space. In this paper we will look at the various kinds and discuss some of the reliability issues involved. In the first half of the paper we will concentrate on the causes of unreliability, illustrating these with some general solutions and examples. Among the issues treated are interprocess communication, machine crashes, server redundancy, and data integrity. In the second half of the paper, we will examine one distributed operating system, Amoeba, to see how reliability issues have been handled in at least one real system, and how the pieces fit together.

1. Introduction

It is difficult to get two computer scientists to agree on what a distributed system is. Rather than attempt to formulate a watertight definition, which is probably impossible anyway, we will divide these systems into three broad categories:

- Closely coupled systems
- Loosely coupled systems
- Barely coupled systems

The key issue that distinguishes these systems is the grain of computation, which can be roughly expressed as the computation time divided by the communication time. If this ratio is below 10, we have a closely coupled system. If it is between 10 and 100 we have a loosely coupled system. Above 100 the system is barely coupled.

In practice, the amount of time required for communication is determined by the communication hardware and the operating system. In a system

* This work was supported in part by the Netherlands Organization for the Advancement of Pure Research (Z.W.O.)

A. S. Tanenbaum and R. van Renesse
Reliability Issues in Distributed Operating Systems
Proc. 6th Symp. Reliability of Distr. Softw. & Datab. Syst.
Williamsburg, Virginia
pp. 3-11
March 1987

consisting on a large number of CPU boards on a single backplane with shared memory, it may be possible for one processor to write a word in another processor's memory in microseconds. On the other hand, processors that communicate over a local area network by message passing typically require milliseconds to send a message and get a reply. Finally, when a wide-area network is being used, communication times of hundreds of milliseconds or more are normal.

These hardware parameters tend to give rise to three kinds of distributed systems, each with their own properties. These systems differ in terms of how the users view the system, how much autonomy the individual processors have, how problems are partitioned among the processors, how work migrates among the processors, how the load is balanced, how interprocess communication is done, whether the system is homogeneous or heterogeneous, and finally how reliable the total system is and what the failure modes are.

At one extreme we have closely-coupled multiprocessors with shared memory communicating over a backplane type bus with short bursts of computation interleaved with short bursts of communication. This is fine-grained parallelism.

Usually all the processors used in this kind of system are identical and fairly close together (same room). Frequently, all the processors are working together on a single problem. Although the system designers may try to make the presence of multiple processors transparent, with hundreds or thousands of CPUs it may be difficult to keep all the processors busy unless the parallelism is programmed explicitly.

The second kind of system is the loosely coupled system, typically consisting of a number of workstations or personal computers communicating over a local area network. In some systems a rack of processors is present, any or all of which can be dynamically allocated as the need arises. In some cases, the user perceives of the system as a collection of autonomous computers that share a common file server or printer. In other cases, the system looks like a virtual uniprocessor. In other words, to the user, the whole system looks like a traditional multiuser time sharing system, rather than a network of independent machines.

There are two general approaches that can be used in such systems. In the first one, all the machines run the same operating system. In the second one, different machines can run different native operating systems, with a layer of software on top to make them look (more) homogeneous. A general survey on distributed systems is given by Tanenbaum and van Renesse [Tanenbaum85].

The third kind of system consists of (typically large) computers or local area networks connected by a low-bandwidth, wide-area network. These machines are barely connected in the sense that communication costs normally dominate the computation costs. Still, for some applications, such as doing joins in a database system, the amount of computing is so large that the system can be made to appear to the user as a single system, despite the low-bandwidth connection between the pieces.

A key point that is common to all these systems, however, is the question of

whether the parallelism provided by the multiple processors is implicit or explicit. This point has important implications for reliability aspects of the system. If the system looks to the user like a virtual uniprocessor, there is relatively little that can be done about reliability at the user level. The reliability must be handled by the system. On the other hand, if users can explicitly control the parallelism, it is possible for them to use the redundancy to enhance the reliability.

A simple example may make this point clear. Some distributed file systems offer atomic transactions [Lampson81] as a primitive operation. The user can specify that a transaction be started, issue commands to read and write files, and then commit the transaction. The system then either runs the entire transaction to completion, or fails, leaving all the files in their original state. Such a file system may well use multiple processors and multiple disks internally, but there is nothing the users can do to influence the reliability behavior.

Now consider a different example, a system with a rack of processors that can be dynamically allocated to processes upon request. A process can request n processors, set all of them working on the same problem (possibly with different algorithms), and then accept the majority answer when all have reported back. In this system the parallelism is explicit, so the user can decide how much redundancy is required for the problem at hand. The conclusion is that systems with explicit parallelism tend to be more flexible, but require more work on the part of the user.

2. Causes of unreliability

Space limitations prevent us from examining the reliability aspects of all three kinds of systems in detail, so we will focus primarily on the middle category—loosely coupled systems. In particular, in this section we will look at some problems that cause systems to be unreliable and on some of the solutions that have been proposed for these problems. In the next section we will look at one distributed system, Amoeba, to see how th ese problems have been attacked and how the various components fit together to make a more reliable system.

2.1. Interprocess communication

When the processors in a distributed system are connected by a "thin wire" local network, interprocess communication primitives that explicitly or implicitly require shared memory (such as semaphores), are not desirable.

Instead some form of message passing is needed. One widely discussed framework for message-passing in computer networks is the ISO-OSI model [Zimmermann80]. To make a long story short, the various protocols that go with this model are so complex and cumbersome, that their use in a high performance local-area network is unattractive at best.

The model favored by most researchers in this area is the client-server model, in which a client wanting some service (e.g., a block from a file) sends a message to the server, which then sends a reply. The basic primitives in the simplest form of client-server model are SEND and RECEIVE, each specifying

an address (destination or source), and a buffer.

These primitives come in several varieties. First of all, there is the question of whether transmission is reliable or not. On some systems, SEND means put the message out onto the network and hope for the best. Processes needing better reliability than that must arrange for it themselves. Other systems use low-level protocols that do automatic timeout and retransmission. Here we see a clear tradeoff between performance and reliability.

A second question is blocking vs. nonblocking primitives. With a blocking SEND, the sender is suspended until the message has been transmitted (unreliable transmission) or transmitted and acknowledged (reliable transmission). With a nonblocking SEND, the sender continues immediately. If the sender modifies the buffer, these changes may or may not be transmitted, depending on whether transmission has taken place or not. Similarly, a blocking RECEIVE waits until a message arrives, but a nonblocking RECEIVE merely provides a buffer. When a message arrives, the receiver gets an interrupt. Nonblocking primitives are harder to use (hence less reliable) but provide more parallelism and higher performance.

Based on experience, many system designer have decided to favor reliability over performance, which has led to the **remote procedure call** [Birrell84, Nelson81, Spector82]. In this scheme, the client makes what looks like a call to a procedure running on the server's machine, but it actually makes a call to a **stub procedure** running on its own machine, as shown in figure 1. The stub procedure packages all the parameters in a message, which it then reliably sends to a stub on the server's machine. The server stub then indeed makes a local procedure call on the server.

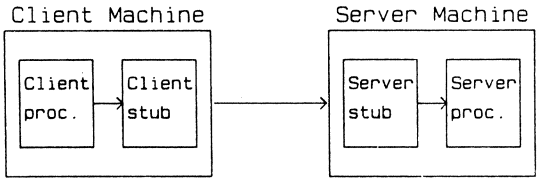


FIGURE 1. Client-Server model.

This model is attractive in many ways. For one thing, the client need not know anything about the fact that the server is remote. It just makes an ordinary procedure call, with the parameters passed in the usual way (e.g., on the stack). Similarly, the server is called by a local procedure according to the local calling and parameter passing conventions. For another thing, the semantics are straightforward and familiar. Programmers understand the procedure call model much better than the message model.

For all its elegance, however, a number of problems lurk under the surface. Many of these have important implications for the system's reliability. Most of them are directly related to the goal of the remote procedure call-transparency, that is, making it look like a local procedure call.

To begin with, when a program makes a local procedure call, the procedure

is executed exactly once, no more and no less. With remote procedure calls, this ideal is unachievable in general. The problem is that the remote server may crash just before or after performing the remote operation, but before sending back the acknowledgement. If the client repeats the request, and it was already carried out, then it will be carried out a second time.

Operations that can be carried out multiple times without harm, such as overwriting a specific disk block are said to be **idempotent**. Unfortunately, most operations that involve communication or I/O are not idempotent. For example, if the request was to a bank server to transfer a large amount of money to a numbered Swiss bank account, one would prefer that operation not be executed by accident a second time.

At first glance you might think that the problem could be solved by having the server record the fact that it was about to perform the operation in a secure way, for example, on stable storage [Lampson81]. However, this idea does not work for nonidempotent operations because after recording its intentions the server has to carry out the operation and then send the acknowledgement. In the best case, each of these steps can be done in a single instruction, for example, by setting one bit somewhere. If the server crashes between the two instructions, when it reboots it cannot determine if the crash occurred just before, between, or just after the two instructions.

This observation leads to three classes of remote procedure call systems: those that have "at least once" semantics, those that have "at most once semantics" and those that have "don't know" semantics. In the former class, if the client stub does not get a reply within a specified interval, it just keeps repeating the request until it gets one. The call may be repeated several times, however.

The second kind of semantics is "at most once." One way to implement this is to simply avoid all retransmissions, but then a simple lost message results in a failed execution. A better way is to have the server log all actions *before* performing them, so that if a repeated request comes in, it can be recognized as such and rejected. With this model, the client knows that the call has been performed either 0 or 1 times, but no more.

The third category consists of systems that give no guarantee at all. These have the advantage of being easy to implement.

Transparency also brings other problems with it. Suppose a server is overloaded. A client that does not realize that the lack of response is due to overload may think it is due to lost messages and keep retransmitting, thus making the problem worse.

2.2. Server crashes

Another source of unreliable behavior is machine failures, either due to hardware or software. These can be split into two categories: server crashes and client crashes. These have different consequences for the system and must be attacked differently. In this section we will look at the problems associated with server crashes and in the next one client crashes.

In general, servers can crash. Obviously one should try to make the servers

as reliable as possible, but even perfect software will not act properly if the hardware refuses to work. Furthermore, making the software perfect is easier said than done. This problem can be approached two ways. One way is to try to get crashed servers back on the air as fast as possible. The other way is to provide multiple servers for redundancy.

Getting crashed servers back up again requires some mechanism to detect when a server has gone done and some way to get it back. Ideally there should also be some mechanism to adjudicate disputes. If the server claims to be up but its clients claim that it is not doing anything, what then? Whatever mechanism is chosen to monitor servers should itself be highly reliable of course.

When the problem of unreliable servers is tackled by having several of each kind, the issue of client-server binding arises. In a system with multiple identical servers (e.g., 3 file servers), at some point a choice must be made about which one a client will use. One can easily imagine a system in which the servers share a common address or mailbox, with each server taking new work out of the mailbox whenever the server is idle. Suppose server 1 takes a request out of the mailbox, carries it out, and then sends an acknowledgement that is subsequently lost.

At this point server 1 crashes. The client times out and retransmits the request, only to have it be taken by server 2 this time, which knows nothing about what server 1 has done recently, because server 1 is currently down and cannot tell it. Server 2 now repeats the request. If the semantics are "at most once" we have a problem. This problem occurs even if server 1 has carefully logged the request and reply in order to filter out repeats.

The difficulty is that the binding between the client and server was automatically broken and reset when the first server went down. Many systems regard automatic rebinding as a step towards fault-tolerant, reliable systems, but we see here that one must be careful.

Another issue related to automatic rebinding of servers is that of state. Some servers may have a long term state that is maintained even after a remote procedure call has terminated successfully. For example, some file servers have an operation OPEN on a file that returns a file descriptor for use in subsequent READs and WRITEs. If multiple instances of such a server exist, problems will arise if the server holding a particular client's open file table crashes between two remote procedure calls so that subsequent calls go to a new server not having the necessary state.

Of course the system can have a rule that odd-numbered clients always use server 1 and even-numbered clients always use server 2, but such a scheme completely defeats one of the goals of a distributed system, namely, to use redundancy to improve reliability.

Yet another reliability problem associated with binding is authentication. How can the server tell which client sent the message, and how can the client be sure he is sending his data to the real server and not to an imposter? Going through a full authentication protocol, complete with passwords, on every call is not feasible. On the other hand, solutions such as that of Birrell

[Birrell85] effectively require setting up a long-term encrypted session, thus moving away from the idea of transparency, since now remote procedure calls need to first set up sessions between client and server, but local ones do not.

2.3. Client Crashes

So far we have only looked at the reliability problems caused by server crashes. Client crashes also cause plenty of headaches. When a client starts up a computation on a server and then crashes, the computation continues even though nobody is interested in it any more. Such a computation is called an **orphan**. Having a lot of orphans lying around making random computations does not enhance the reliability of a system. Orphans are most serious when the computation being done by the server takes a substantial amount of time.

Various methods, some fairly draconian, have been proposed for dealing with orphans. One method is to kill off all processes in the whole system every T seconds. This will certainly kill off all the orphans, but it is something of a nuisance to normal computations.

Another possibility is to have each server periodically check to see if the client that started the current computation is still interested. A variation on this idea is the **dead man's handle**. A client is expected to poll a server working for it periodically. If a poll fails to come in on schedule, the server just kills the computation.

A different approach is to program all clients to log all remote procedure calls on stable storage before making them. When a client reboots after a crash, it checks to see if there were any servers working for it, and if so, tells them to stop. This solution is expensive because writing to disk to log each call doubles the cost of each remote procedure call.

No matter which of these methods is chosen for killing off orphans, there is always the danger than an orphan will be in the middle of a critical section at the instant that it is killed, or that it holds many locks on resources. In this case, killing the orphan can lead to race conditions and deadlocks.

Even if a method can be found to kill off all orphans, it may well be that an orphan has created some long term state that will cause other actions to happen later. For example, a file may have been put in a queue for subsequent processing elsewhere in the system. Thus even after an orphan has been killed off, some other processor may examine the queue, find the work, and start up another orphan.

Let us now briefly look at some systems that have attempted to deal with server and client crashes. Borg et al [Borg83]. have described a system in which each process has a backup process running on a different processor. Whenever a client sends a message to a server, it also sends the same message to the server's backup, as shown in figure 2. Similarly, replies are sent to both the client and its backup. The operating system takes care of coordinating and synchronizing all the messages.

The idea behind this technique is that if a process crashes, its backup, on another processor, will be available to take over. Of course this scheme

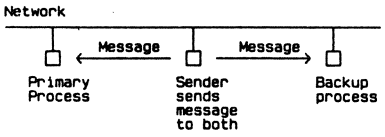


FIGURE 2. Each process has its own backup.

requires doubling the number of processors. Powell and Presotto [Powell83] have proposed a simpler scheme that only requires one extra process, instead of doubling the number of processes. In their scheme, shown in figure 3, there is a single **recorder process** that logs all messages sent on the network.

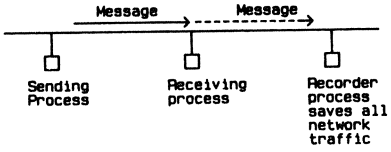


FIGURE 3. A recorder process logs all message traffic.

If a process crashes, a new processor can be allocated, and the code of the crashed process loaded into it. Then the recorder carefully spoon feeds the new process all the messages it has saved, in order to get the new process into the same state as the old one was when it went down. Messages sent by the process while it is getting to the point where the old one was are intercepted just before they are sent, to prevent their recipients from being confused. When the new process gets to the point that the old one was, it switches into normal mode, so that messages really are sent.

Processes can also make checkpoints of themselves from time to time if they wish. Doing so means that if a process crashes, the checkpoint can be started up and only messages logged after the checkpoint was made have to be replayed.

Powell and Presotto's technique has the advantage of not requiring any overhead during normal operation. However, it does implicitly presume that all messages are correctly received and logged by the recorder.

A different approach to reliability is Cooper's [Cooper85] replicated proced ure call. In Cooper's model, each client process is in reality n processes running in parallel and executing the same code. Similarly, each server consists of m parallel processes. When a client calls a server, each client process sends a message to each server process.

When the replies come back to the client, they are compared. One possible comparison algorithm is to vote. Whichever answer occurs the most times is declared the winner, and given to each client. The clients then continue their work. In this manner, an occasional error is simply voted down, thus giving a

degree of fault tolerance.

2.4. Data Integrity

Another key reliability issue is data availability and integrity. If data are frequently inaccessible because some key server is down, users will perceive the system as unreliable. This problem can be dealt with to some extent by having multiple servers of each type, each holding its own private copy of the data. As long as the data are never changed (or very rarely changed), this solution works well. However, if updates are frequent, the redundancy itself introduces problems.

The main problem, of course, is that having multiple copies of the data introduces the possibility of the various copies becoming different over the course of time. Before looking at the replication problem, let us first take a look at the good old days of magnetic tape. In those days, it was common for companies to have a master tape with their current inventory of products. Each day tapes containing the day's purchases and sales would be brought to the computer center. The master tape, an update tape, and a blank tape would be mounted, and a job run making an updated master on the blank tape. Then the next update tape would be run with the new master, and so on.

The nice thing about this system was that if the computer crashed at any instant, it was always possible to go back to the original or any other master tape and start everything again. When magnetic disks were introduced, systems began updating records in place, losing the idempotency of the tape scheme. Furthermore, when multiple update runs were allowed at the same time, sophisticated concurrency control algorithms had to be introduced to make the updates serializable while avoiding deadlock. In this view, the very concept of updating files in place on the disk is seen as a major source of unreliability. When the situation is further complicated by having the work distributed over multiple machines, the potential reliability problems become even worse.

Assuming the problems of concurrency control and serializability on a single machine can be dealt with by conventional means, the issue of replication can be dealt with in several ways. The first way is to have a master copy with multiple backups. This scheme closely resembles the old tape system. After the master copy has been updated, the changes have to be propagated to the backups.

The second way is to update all the copies in parallel, but when inconsistencies arise, to vote [Thomas79, Gifford79]. In this way minority viewpoints can be stamped out.

A third scheme is regeneration [Pu86]. When an update is done, the server doing the update arranges for multiple copies to be made. If one of those subsequently becomes disconnected or unavailable, the server just abandons the missing copy and generates a new one.

3. Reliability in Amoeba

In this section we will look at the Amoeba distributed operating system [Mullender84, Mullender85, Mullender86, Tanenbaum86] to see how reliability issues have been dealt with in a real system. First we give a brief introduction to Amoeba.

Amoeba is a distributed operating system that has been designed and implemented at the Vrije Universiteit and the Centrum voor Wiskunde en Informatica. It runs on a collection of 40 Motorola 68000s, 68010s, and 68020s connected by a 10 Mbps local area network. The conceptual model behind the system is the abstract data type. Client processes can perform operations on objects managed by servers. These operations are implemented by having the clients send messages to the servers, with the servers sending the results of the operations back to the clients. This is a simple form of remote procedure call.

Both client and server processes, called *clusters*, can consist of multiple *tasks* that conceptually run in parallel within the same address space. While one task is blocked waiting for a message, another one can be running. Many servers are implemented as a collection of tasks, each of which starts out waiting for a message. When a request to perform an operation arrives, it is given to one of the tasks at random. If that task should later block (e.g., waiting for a disk), another task in the cluster can run on behalf of a different client. Synchronization is achieved by never switching from one task to a different task in the same cluster except when the current task is logically blocked. The scheduler can switch between clusters at will, however.

The Amoeba system consists of four basic components, as shown in figure 4. The workstations are used to provide a multi-window interface to the user, as well as some local computing such as editing. The pool processors can be dynamically allocated as needed for compilations, text formatting, or doing any other work. An *n-pass* compiler, for example, can be arranged to allocate, use, and then return *n* pool processors, one per pass.

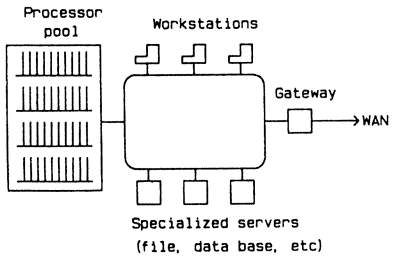


FIGURE 4. An Amoeba system has four components.

The system also contains specialized servers with dedicated functions, such file servers, bank servers, and boot servers. Finally, the fourth component is the gateway to other Amoeba systems. Soon Amoeba will be running at five

sites in three countries, all interconnected by a wide-area network.

Identical Amoeba kernels run on all the machines. The kernels are intentionally small, basically handling only communication and low level memory management. Files, process management, and even protection and accounting are all handled at the user level.

Objects are protected by capabilities, as shown in figure 5. Each capability contains a *port* field that is used to identify the server or client being addressed and an object field, used to identify the specific object to be manipulated. Object numbers are analogous to i-node numbers in UNIX.† Next comes a rights field, telling which operations the holder of the capability may perform on the object. Finally, there is a random number that prevents users from forging capabilities. Capabilities are directly handled by user processes, outside the kernel.

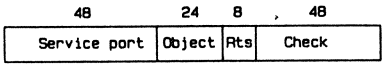


FIGURE 5. An Amoeba capability.

The random number field is crucial to the protection scheme, hence to the reliability of the system. When an object is created, the creating server allocates an "i-node" for it and puts a random number in it. It then EXCLUSIVE ORs the rights bits (initially all 1s) with the random number and runs the result through a one-way function [Evans74] used for all objects. The output of the one way function is put into the random field of the capability. The rights bits are included in the capability in plaintext.

When a client performs an operation on an object, the capability for the object is sent to the server to identify the object. The server then uses the object number contained in the capability as an index into its tables to find the random number. The random number thus found is EXCLUSIVE ORed with the plaintext rights field and run through the one-way function. If the output is the same as the capability's random number, the capability (including the plaintext rights bits) is accepted as valid. This protection system and several variations on it are described in more detail in Tanenbaum et al [Tanenbaum86].

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

3.1. Interprocess communication

The form of remote procedure call used by Amoeba has "at most once" semantics. For most applications this is preferable to "at least once" and certainly better than "don't know." We will now describe how these semantics are implemented.

When a remote procedure call is made, the client calls a stub procedure that locates a server based on the port number present in the capability belonging to the object to be operated upon. The location is done by first looking in a cache. If that fails, a broadcast is done. If multiple servers handle the object class in question, the stub selects one of them, and gets its process identifier (pid).

Then a message is sent to the selected server process. Normally, the server will perform the operation and send back a reply. If the server's reply is not forthcoming within a certain time interval, the server's stub times out and acknowledges receipt of the request so the client will know that it arrived safely and that the server is hard at work on it. When the server's reply finally gets back to the client, the client's stub sends an acknowledgement back to the server, which terminates the call.

If it has received an acknowledgement but no reply to the request itself, at a certain point the client gets nervous and sends an "Are you alive?" query to the server, which is answered immediately. On the other hand, if the client has heard nothing at all from the server, not even the acknowledgement of the request, it eventually times out and retransmits the request. When the server sees the retransmitted request, which bears the same source and request number as the original, it can recognize the request as a retransmission and just send the reply again or at least just acknowledge receipt of the request if the result is not yet available.

Now consider what happens if the server crashes. The client stub eventually detects that the server process is down when it fails to get answers to its "Are you alive?" messages. If the client stub has enough knowledge of the specific operation to be sure that it is idempotent, it can locate another server and repeat the operation. In this case it does not matter that the operation was executed more than once.

On the other hand, if the stub does not know whether or not the operation is idempotent, it simply reports back failure to the client, meaning that that the operation has been performed either 0 or 1 times, but not more.

3.2. Server crashes

The communication mechanism is not the only part of Amoeba that was designed with reliability in mind. There is also a *boot server* whose job is to make sure that processes (typically servers) that are supposed to be alive are in fact alive. It does this by periodically probing the registered servers to see if they are still functioning.

All the long-lived servers, such as the file servers, normally register with the boot server when the system comes up. This registration consists of providing the boot server with the message to be sent to the server and the reply that the

server is supposed to send back, the frequency at which these probes are to take place, the number of probes to make before declaring the server dead, and the procedure for creating a new server to replace one that has crashed.

The procedure used to reincarnate a crashed server depends on the nature of the crash. If the server is dead but the kernel on its machine is still working, then the boot server instructs the kernel to create a new server process to replace the old one.

If the entire machine has crashed, then the boot server sends a special packet on the network that is detected by the interface card, and which results in the interface asserting a RESET signal on the crashed machine's bus. This signal causes the machine to reboot itself by jumping to a program in a ROM. The ROM program and the boot server together download a new kernel into the machine, at which time the server can be restarted. If the machine cannot be started up at all, the boot server gets another processor and starts the server there. This whole procedure is fully automated; it happens without human intervention.

The only other issue concerning the boot server is the reliability of the boot server itself. Multiple copies of the boot server run, each one communicating with all the other ones. If one of the boot servers crashes, the remaining ones regenerate it using the procedure just described.

3.3. Client crashes

Orphans are prevented in Amoeba by using the "Are you alive" messages as a dead man's handle. If a server is making a long computation, it expects to get "Are you alive messages" periodically. If these messages cease to arrive, the server concludes that the client is dead and kills the orphan itself.

Although the orphan detection mechanism is useful for ridding the system of unwanted computations, in many circumstances it is desirable that clients be fully fault tolerant, meaning that a client, especially one running in parallel on multiple pool processors, itself notices crashes of some of its processors and recovers from them in a transparent way. Several applications have been programmed in this way. Below we will briefly sketch two of these, the traveling salesman problem and parallel alpha-beta search.

The traveling salesman problem consists of finding the shortest route that a salesman can use to visit all the cities in his territory exactly once. Roughly speaking, the Amoeba approach is to have a procedure, *traverse*, that takes as input a partial path, the set of cities as yet unvisited, and the length of the best total path found so far [Bal85]. This procedure forks off a process for each unvisited city to investigate all paths with that city as the next step. Each process simply runs *traverse*, with a partial path one city longer and the set of unvisited cities one smaller. The recursive forking of parallel processes continues until a certain depth in the tree has been attained, at which point the residual tree is searched completely by one process. Variations of this search strategy have also been tried.

The reliability comes from the fact that if a process fails to report back its findings within a certain time, and also fails to respond to the "Are you alive"

messages, the process that invoked it just asks for another pool processor and starts the work all over again. Higher levels in the tree do not even know that a fault has been detected and corrected. In this way the program will be executed correctly even in the face of repeated multiple processor crashes.

The other reliable application that has been tested is heuristic search for the game reversi (Othello) using the alpha-beta algorithm. At each board position a process is generated for each legal move. Although the details of alpha-beta make this application somewhat different than the branch and bound algorithm used for the traveling salesman, again if a process crashes, its parent just finds someone else to do the work. As we mentioned in the introduction, the fact that the parallelism is visible to the application makes it possible to exploit it for better reliability.

3.4. Data integrity

File servers in Amoeba are user-level processes, so there can be several of them running at once, providing different services and serving different clients. Some of the file servers have been designed to provide UNIX-file service, others have been designed for high performance, but there is also one whose goal is high reliability. This one, called FUSS (Free University Storage System) is described by Tanenbaum and Mullender [Mullender85] and is sketched below.

The technique used by FUSS to provide high reliability is the *immutable file*. When a process wants to update a file, it asks FUSS to create a new version of the file and return a capability for the copy. (Actually the file is not copied. Shadow pages are used, but this is really just an optimization.) The process can then modify the copy as it wishes. When it is done, the process tells FUSS to *commit* the file, making the copy the new file. Thus a file is really a sequence of versions, none of which is ever modified once it has been committed. Modifying a file consists of atomically replacing a file with a new version.

This design is more reliable than the traditional update-in-place file system because updating a file consists of preparing the new file and then at the last minute switching one pointer. If the file server crashes, either the old file or the new file will be present when it comes up again, but never a mixture of the two. By appropriate logging of intentions on a disk, the server can be made to eventually complete the update no matter how often it crashes. The atomic update property is especially important if two or more processes are simultaneously updating the same file. FUSS offers a choice between locking and optimistic currency control, but in both cases, an update to a file (or even a set of files) is atomic.

Work is currently in progress to extend these ideas to general objects. The idea is that any object should be representable as a sequence of versions, with the update from the old version to the new one being done atomically. This can be achieved by having a directory server that maps ASCII strings onto capabilities, or more generally, onto sets of capabilities. In effect, a directory is an unordered collection of lines, each containing a ASCII object name followed by set of capabilities. The capabilities are for replicas of the same object.

A directory is thus simply a mapping of ASCII names onto sets of objects. A directory is itself an object, so directories can contain capabilities for other directories, giving rise to a directory hierarchy, or even a general graph, if that is desired.

The principal operation on a directory object is to present the directory server with a capability for a directory and an ASCII string to be looked up in that directory. The server then looks up the given string in the directory and returns the full set of capabilities that correspond to that string, if any. The client can then choose one of them at random to use. If that one is not available, it can choose another one.

The idea of having the directory entry contain multiple capabilities has been done to enhance the reliability. Because files (and objects generally) are immutable, once a new version of an object has been created, the directory server can arrange for backup copies of the object to be made at its leisure (lazy backup). There is no problem with race conditions because the object cannot change. The worst that can happen is that the version being backed up becomes obsolete before all the backups have been created, in which case some extra work may have been done for nothing, but the file system integrity is never affected.

Updating a directory entry is done by sending the directory server a capability for a directory, an ASCII string, the capability for the object being replaced, the capability for the new object, and a count specifying how many backup copies should be made and maintained. The directory entry is updated atomically—either it happens or it fails, but there is never half an update. Notice that the replication effort is managed by the directory server, so it need not be duplicated in each object server. This is possible because objects are immutable. Once an object has been committed, it never changes; it can only be replaced in its entirety by a new object.

The update operation requires the old capability as a parameter so the directory server can verify that the object being replaced is still the current object. If the old capability is not present in the set of capabilities for the given string, the directory server can see that another update has transpired in the meantime, so the update operation fails. This scheme is a form of optimistic concurrency control. Put in other terms, if two clients each look up a given string in a given directory, and then both try updating the corresponding object, only the first update will succeed. Objects can also be locked, to allow a more conventional update strategy.

3.5. Other reliability features of Amoeba

Another area that affects system reliability is resource management. If one user or process consumes too many resources, the rest of the users and processes will suffer the consequences. For this reason Amoeba has a *bank server* that can be used as a general tool for resource management.

The bank server manages bank accounts in various currencies. As an example of its use, consider a file server that wished to implement a quota system to give each user at most 1000 disk blocks. Each user would be given a bank

account containing, say, 1000 zlotys, each good for one disk block. Every time a user wanted another disk block, he would first have to transfer 1 zloty to the file server's account to pay for it in advance. When the block was freed, the user would get his zloty back.

Other currencies can be used for other resources. CPU time could be charged in yen, phototypesetter pages in guilders, etc. The *policies* (e.g., who gets how much money, whether currencies are convertible) are decided by the servers, but the basic *mechanism* (managing the accounts, logging transactions, transferring money between accounts atomically, maintaining caches for efficiency, etc.) is done by the bank server, so that each individual server need not run its own administration.

Try as we may to build a reliable system, there are going to be bugs in it. For this reason, Amoeba has been designed in such a way to be able to catch faults and handle them. To see how this mechanism works, we have to take a look at how processes are managed in Amoeba. When a user types a command to the shell, the shell creates a *mother* process to oversee the execution of the command. The mother process allocates a processor from the processor pool, asks the Amoeba kernel on that machine to allocate sufficient memory for the new process, and then downloads the program to be executed to the processor for execution.

Normally, the mother process does not intervene in the execution of the program on the pool processor. It simply waits until the program terminates to clean it up and report back its status. However, it is possible to tell the pool processor's kernel to catch all system calls and other kernel traps, and send them to the mother process for processing.

In this way, for example, it is possible to take a *binary* program compiled to run on 68000 UNIX (i.e., not on Amoeba) and run it on a pool processor, even though the Amoeba kernel knows nothing at all about UNIX. The UNIX system calls are effectively all passed to the mother process for execution. If the mother process happens to be running on a 68000 UNIX system (which is easy to arrange), it can just execute the system calls locally and send back the results.

This same mechanism is used for debugging. When a process on a pool processor gets a memory fault, illegal instruction, or other kernel trap, the pool processor's kernel does a remote procedure call with the mother process telling it what happened. The mother process contains a debugger that can print a message on the user's terminal and then wait for input instructing it what to do. There are commands to examine and print memory and so on. These are handled by messages between the mother process and the kernel on the pool processor.

SUMMARY

Reliability considerations have influenced the Amoeba design in a number of ways. These include the scheme for protecting objects with cryptographically secure capabilities, the communication mechanism with "at most once" semantics and orphan extermination, the boot server for automatically rebooting dead processes, the file server with immutable files, the directory sever with atomic update on replicated objects, the bank server for limiting resource usage, and the hooks for debugging. In addition, Amoeba has been used for explicitly programming fault tolerant applications such as the traveling salesman and heuristic search.

REFERENCES

[Bal85]
BAL, H.E., RENESSE, R. VAN, and TANENBAUM, A.S., "A Distributed, Parallel, Fault Tolerant Computing System", VU Informatic Rapport nr. IR-106, Vrije Universiteit, Amsterdam, October 1985.

[Birrell84]
BIRRELL, A. D. and NELSON, B. J., "Implementing Remote Procedure Calls," *ACM Trans. Comp. Syst.*, vol. 2, pp.39-59, Februari 1984.

[Birrell85]
BIRRELL, A. D., "Secure Communication Using Remote Procedure Calls," *ACM Trans. Comput. Syst.*, vol. 3, pp.1-14, Feb. 1985.

[Borg83]
BORG, A., BAUMBACH, J., and GLAZER, S., "A Message System Supporting Fault Tolerance," *Proc. Ninth Symp. Operating Syst. Prin.*, pp.90-99, 1983, ACM.

[Cooper85]
COOPER, E. C., "Replicated Distributed Programs," *Proc. 10th ACM Symp. on Operating System Principles*, pp.63-78, December 1985.

[Evans74]
EVANS, A., KANTROWITZ, W., and WEISS, E., "A User Authentication Scheme Not Requiring Secrecy in the Computer," *Comm. ACM*, vol. 17, no. 8, pp.437-442, August 1974.

[Gifford79]
GIFFORD, D. K., "Weighted Voting for Replicated Data," *Proc. 7th Symp. on Operating System Principles*, 1979.

[Lampson81]
LAMPSON, B. W., "Atomic Transactions," pp. 246-265 in Distributed Systems — Architecture and Implementation, Springer-Verlag,, Berlin (1981).

[Mullender84]
MULLENDER, S. J. and TANENBAUM, A. S., "Protection and Resource Control in Distributed Operating Systems," *Computer Networks*, vol. 8, no. 5,6, pp.421-432, 1984.

[Mullender85]
MULLENDER, S. J. and TANENBAUM, A. S., "A Distributed File Service Based on Optimistic Concurrency Control," *Proceedings of the 10th Symposium on Operating Systems Principles*, pp.51-62, December 1985.

[Mullender86]
MULLENDER, S. J. and TANENBAUM, A. S., "The Design of a Capability-Based Distributed Operating System," *The Computer Journal*, vol. 29, no. 4, pp.289-300, 1986.

[Nelson81]
NELSON, B. J., "Remote Procedure Call", Ph.D. dissertation CMU-CS-81-119, Carnegie-Mellon University, 1981.

[Powell83]
POWELL, M. L. and PRESOTTO, D. L., "Publishing-A Reliable Broadcast Communication Mechanism," *Proc. Ninth Symp. Operating Syst. Prin.*, pp.100-109, 1983, ACM.

[Pu86]
PU, C., NOE, J. D., and PROUDFOOT, A., "Regeneration of Replicated Objects: A Technique and its Eden Implementation," *Proc. Second Int'l Conf. on Data Engineering*, pp.175-187, Feb. 1986.

[Spector82]
SPECTOR, A. Z., "Performing Remote Operations Efficiently on a Local Computer Network," *Comm. ACM*, vol. 25, no. 4, pp.246-260, April 1982.

[Tanenbaum85]
TANENBAUM, A. S. and RENESSE, R. VAN, "Distributed Operating Systems," *ACM Computing Surveys*, vol. 17, no. 4, pp.419-470, December 1985.

[Tanenbaum86]
TANENBAUM, A. S., MULLENDER, S. J., and RENESSE, R. VAN, "Using Sparse Capabilities in a Distributed Operating System," *Proc. of the 6th Int. Conf. on Distributed Computing Systems*, pp.558-563, May 1986, Vrije Universiteit.

[Thomas79]
THOMAS, R. H., "A Majority Consensus Approach to Concurrency Control," *ACM Trans. on Database Systems*, vol. 4, pp.180-209, June 1979.

[Zimmermann80]
ZIMMERMANN, H., "OSI Reference Model—The ISO Model of Architecture for Open Systems Intercon nection," *IEEE Trans. Comm.*, vol. COM-28, pp.425-432, April 1980.

File System

A Distributed File Service Based on Optimistic Concurrency Control

Sape J. Mullender
Centre for Mathematics and Computer Science
Amsterdam, The Netherlands

Andrew S. Tanenbaum
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Principles are presented for a distributed file and database system that leaves a large degree of freedom to the users of the system. It can be used as an efficient storage medium for files, but also as a basis for a distributed data base system. An optimistic concurrency control mechanism, based on the simultaneous existence of several versions of a file or data base is used. Each version provides to the client that owns it, a consistent view of the contents of the file at the time of the version's creation. We show how this mechanism works, how it can be implemented and how serialisability of concurrent access is enforced. A garbage collector that runs independent of, and in parallel with, the operation of the system is also presented.

1980 Mathematics Subject Classification: 68A05, 68B20, 68H05
CR Categories: D.4.3, H.2.2, H.2.4, H.3.2.
Keywords & Phrases: file server, data base server, distributed control, optimistic concurrency control, atomic update, serialisability, differential files.

1. Introduction

File systems play an important role in allowing information to be widely accessible, since most information is in some way or another stored on files. There are many different kinds of file systems for distributed systems, 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.

A Distributed File Service Based on Optimistic Concurrency Control
S. J. Mullender and A. S. Tanenbaum
Proceedings of the 10th Symposium on Operating Systems Principles
Orcas Island, Washington.
pp. 51-62
December 1985

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 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. 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 much 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 will be increased by several orders of magnitude.

This paper goes into the design of the distributed file service for the ***Amoeba*** Distributed Operating System [Mullender86]. We have attempted to build a file service, suitable for many different applications: ordinary 'plain' files, hierarchically structured files, replicated files, databases, source code control systems [Rochkind75], etc.

2. Design considerations

Important in the design was the Bauer principle, governing the whole of the design of Amoeba, '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 the 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 the simple-minded user who just wants a reasonably reliable repository for his files, cheap and fast, while on the other hand, the sophisticated user should be taken into account who needs ultra-reliable storage for his files, fancy synchronisation of access by many simultaneous users, and guaranteed availability, who is prepared that it will be expensive and slow.

Another important issue in the design of a file server is that the file server 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 it. 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 often 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 [Mullender85]. We want a free-standing file server, providing disk space for the users of hosts with no, or not enough disk storage of their own.

2.1. File servers in open operating systems

In an open system, several different services may offer the same facilities, albeit in different forms. There can be several file servers, one offering ordinary linear files, another tree structured files with concurrency control mechanisms to arbitrate updates by a number of simultaneous users. The choice of which file server to use is up to the user.

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.

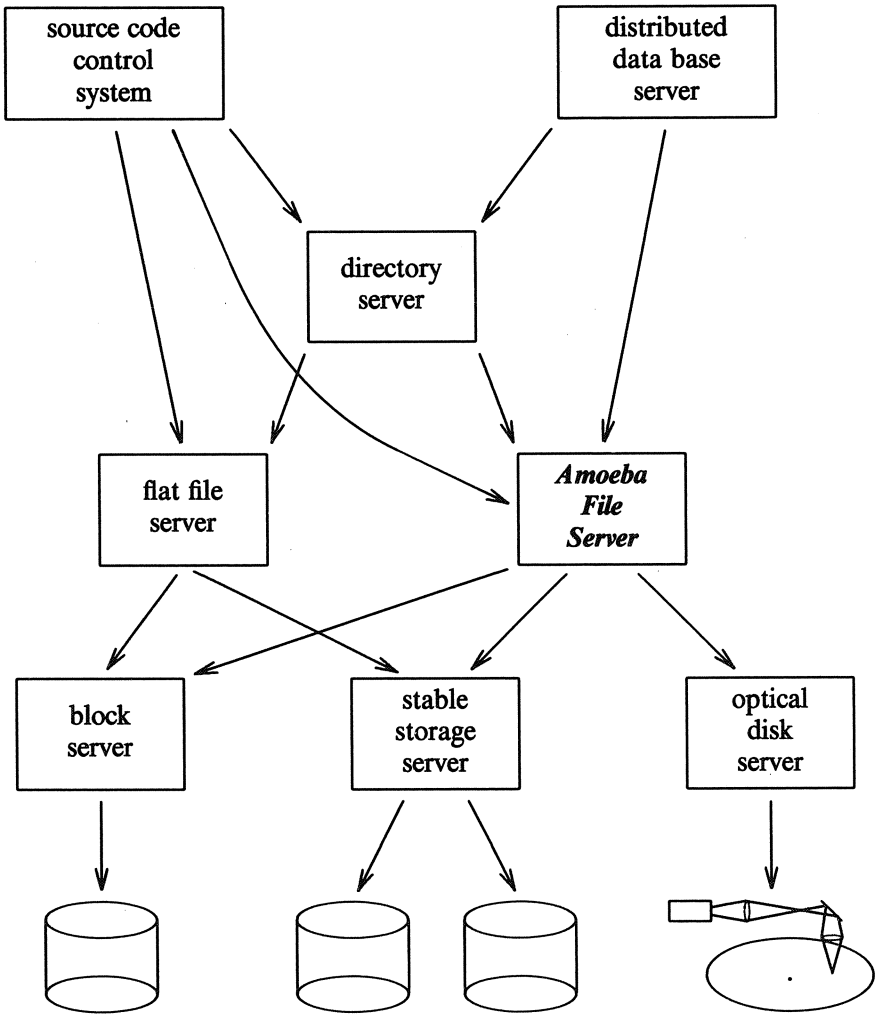


FIGURE 1. An example of a storage services hierarchy in an open system.

Data base management systems often have their own operating systems,

tailored to this particular application, because traditional operating systems provided the wrong functionality [Stonebraker81, Tanenbaum82]. 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 , allows a logical layering of facilities while the development effort can be shared.

The bottom of the hierarchy is formed by the block server, which manages blocks of data of fixed size. At the next level, file services manage filesstructured collections of dataand implement operations for inspecting and changing them. These operations must support the next level, where data, stored in files, is interpreted: the contents of a file may represent the state of an airline reservation system, or the contents of the bank accounts of a branch office, or a pascal program.

File services must provide the tools for the efficient implementation of as wide a set of applications as is 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.

3. 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 [Sturgis80] Felix [Fridrich81] and Swallow [Reed81]. They all have mechanisms for concurrency control. Most file servers, including the Cambridge File Server [Dion80], XDFS and Felix use *locking*[Eswaran76], while some, among them Swallow , use *timestamps*[Reed78].

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.

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 server has locked a datum for some time, a timer expires and the lock becomes *vulnerable*. Another server, waiting on that lock, can then prod the first, 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. The version can be thought of as a copy of the file at the time of its creation, although the file is not actually copied block for block then. Sharing is supported by six access modes. Files are tree-structured. When a new version or a virtual copy is created, the whole tree is initially shared with the most recent version. When it is modified, a copy-on-write mechanism is used, leaving the original

tree intact.

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).

3.1. Advantages over previous file systems

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 [Eswaran76] with an optimistic concurrency control mechanism [Kung81, Robinson82, Schlageter81]. 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 an optimistic approach, is that the file system is always in a consistent state. Most file systems, using other mechanisms for concurrency control, need a mechanism for bringing back the file system to a consistent state after a crash. 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. This is no problem with the Amoeba File Service. The file system is always in a consistent state (assuming the updates themselves are consistent). Server crashes have no serious consequences: the file system is always in a consistent state, so 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 it.

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. 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. 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. The integrity of the cache is 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. The Amoeba File Service needs no unexpected 'unsolicited messages.'

4. The block server

The principle of separating the issues of file service an block service makes it easy to combine different methods of storage (*e.g.*, stable storage [Lampson79]), 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. Based on this, file services can realise concurrency control policies. The Amoeba File Service, for commit on a version of a file, for instance, will lock and read a block, examine and modify it, then write and unlock the block again.

We expect that the block server's clients will often use a small portion of each block for redundancy purposes. Block servers can support a *recovery* operation, which given an account number, returns a list of block numbers owned by that account. A client, *e.g.*, a file server, can then use its redundancy information to restore its file system after a severe crash.

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 [Lampson79] have suggested a method to use dual disk drives to implement *stable storage*. We propose a small modification to their method to make a more reliable version of stable storage.

In our proposed method, each block is stored by two servers on two different disk drives (in contrast to Lampson and Sturgis' method which uses one server and two disk drives). On request to allocate and write a block, the receiving block server, say server *A* allocates a block on its local disk, then sends a request to its companion block server, server *B* including the data and the chosen block number. *B* then writes the block to disk at the address indicated by *A*, and sends an acknowledgement back to *A*. Finally *A* writes the data in its own block, and returns an identifier for the block to the client.

Read and write requests can be sent to either block server. For reads, the block server need not consult its companion server, except when the block on its disk is corrupted. For writes, the same message exchange is used as for allocate and write.

Allocate collisions may occur when two clients allocate a block simultaneously, one on server *A* and one on server *B*, and, accidently, *A* and *B* choose the same block number. Similarly, write collisions may occur when two clients write the same block via different block servers. These collisions are detected, however, before any damage is done, because writes are always carried out on the companion disk first. When a collision is detected the companion server is warned, and appropriate measures can be taken (*e.g.*, redo the operation after a random wait interval).

After a crash, the block server compares notes with its companion, and restores its disk before accepting any requests. To this end, block servers make intentions lists for crashed companion servers. Clients send requests to the alternative block server if the primary fails to respond. Otherwise crashes are dealt with in the same manner as in Lampson and Sturgis' method.

5. Amoeba file service

The Amoeba File Service was developed for, but is not restricted to, the Amoeba Distributed Operating System [Mullender86]. 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 conflict in simultaneous updates.

For concurrency control, three mechanisms stand out as the most frequently used: locking [Menasce], timestamps [Reed78], and optimistic [Kung81]. Each method has advantages and drawbacks, and the discussion which method is best will continue for some time. Several file servers have been implemented with a concurrency control mechanism. Most of these, however, use locking as their concurrency control mechanism [Fridrich81, Sturgis80, Dion80], except a few that use timestamps [Reed81]. File servers that use optimistic concurrency control, however, are not known to us, although, as we shall see, optimistic concurrency control has some properties that make it very attractive for application in a file server.

The Amoeba File Service implements optimistic concurrency control by a version mechanism: When a client modifies a file, a new version of the file must be 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. Several versions of the same file can exist at the same time. The Amoeba File Service checks on commit whether the modifications to the file constitute a serialisability conflict (see [Kung81]).

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.

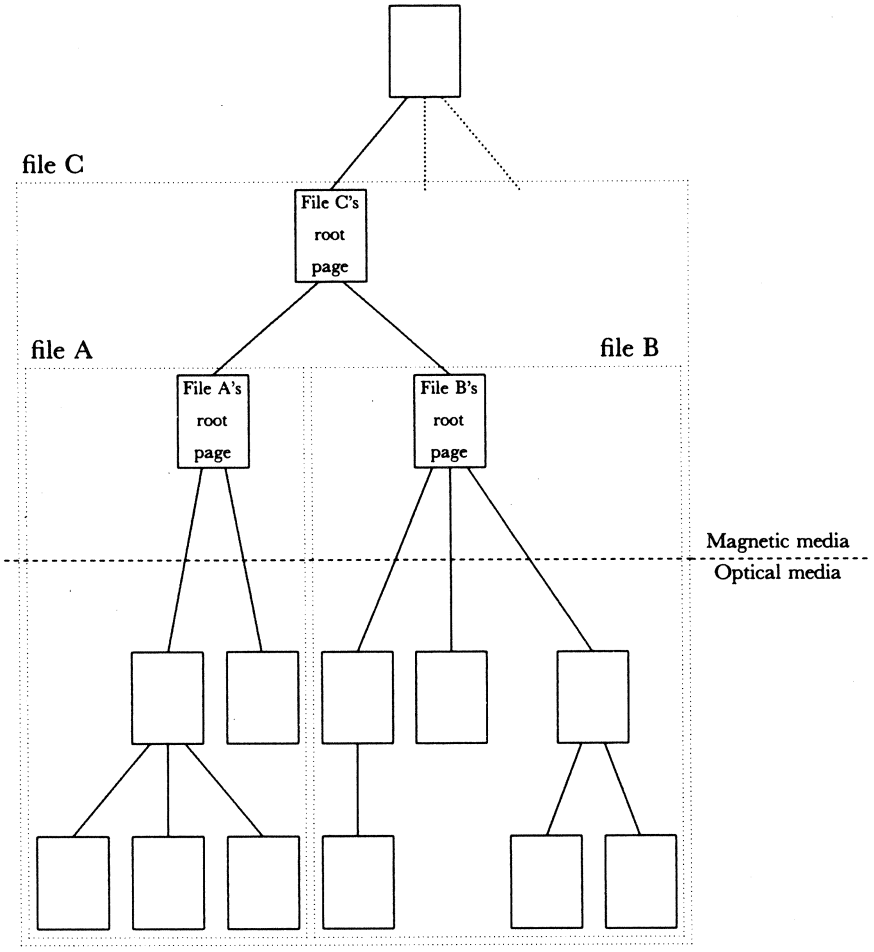


FIGURE 2. The file system has the structure of a tree. Files also, consist of trees of pages. The file system can be viewed as a tree of trees.

The file system as a whole is represented as a large tree of *pages*. The top of the tree (*i.e.*, near the root) is stored on magnetic random-access media, for instance, such as provided by the *stable-storage* server, described in the previous section. The lower parts of the tree can be stored on magnetic disk, or write-once media, such as optical disk. As illustrated in , 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*. For the moment, this hierarchy will be ignored; we shall consider a file system where the upper part of the tree consists of only one page; that is, a file system containing only one file. Later, we shall return to the general situation, where the top part of the page tree forms a 'real' tree.

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 a message in a transaction: 32K bytes. This ensures that pages can be read and written in one (atomic) transaction.* 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. The remaining space in a page can be occupied by references to pages in the next level of the page tree.

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 intent 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.).

5.1. File representation

A file - in this section we should perhaps say '*the* 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 .

* 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.

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 page of a version tree is referred to as the **version page**. The data in a page has no predefined structure. Clients are free to write them as they see fit. The references in a page are for internal use by the Amoeba File Service and can only be read and written by servers.

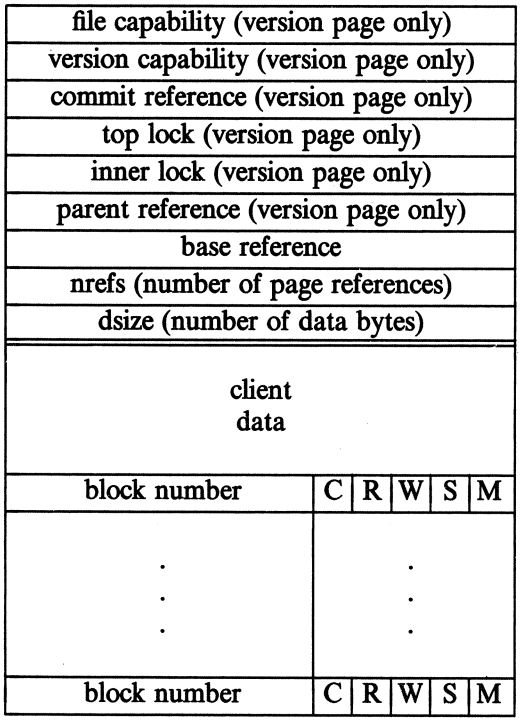


FIGURE 3. The Amoeba File Service page layout

The lay out of a page is shown in figure 3. The page is divided in two areas, the header area and the page itself; the separation is indicated by the double line. The first field in the header area is the *file capability*. This field gives the capability of the file whose root the page is. The next field is the *version capability*, the version of the file whose root the page is. The *commit reference* field is only used in version pages; 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 file whose root is higher in the page tree. In this section we have assumed there is only one file in the system, so these fields are not used here; their function will be explained in a later section. The *parent reference* gives the name of the parent version block. *Parent references* can be used to ascend the upper part of the page tree to the root. The fields mentioned just now are only present in a version page. They are absent (or ignored) in other pages. The *base reference* field 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. 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 section 5), 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 (*insert page*, *remove page*, *make hole*, *remove hole*). 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 references 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. These path names are visible to clients, giving them explicit control over the structure of their files.

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 one in , where we have just shown the version pages and their base and commit references.

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 independent of one another. The *S* flag tells that the references have been referenced, the *M* flag tells whether the references

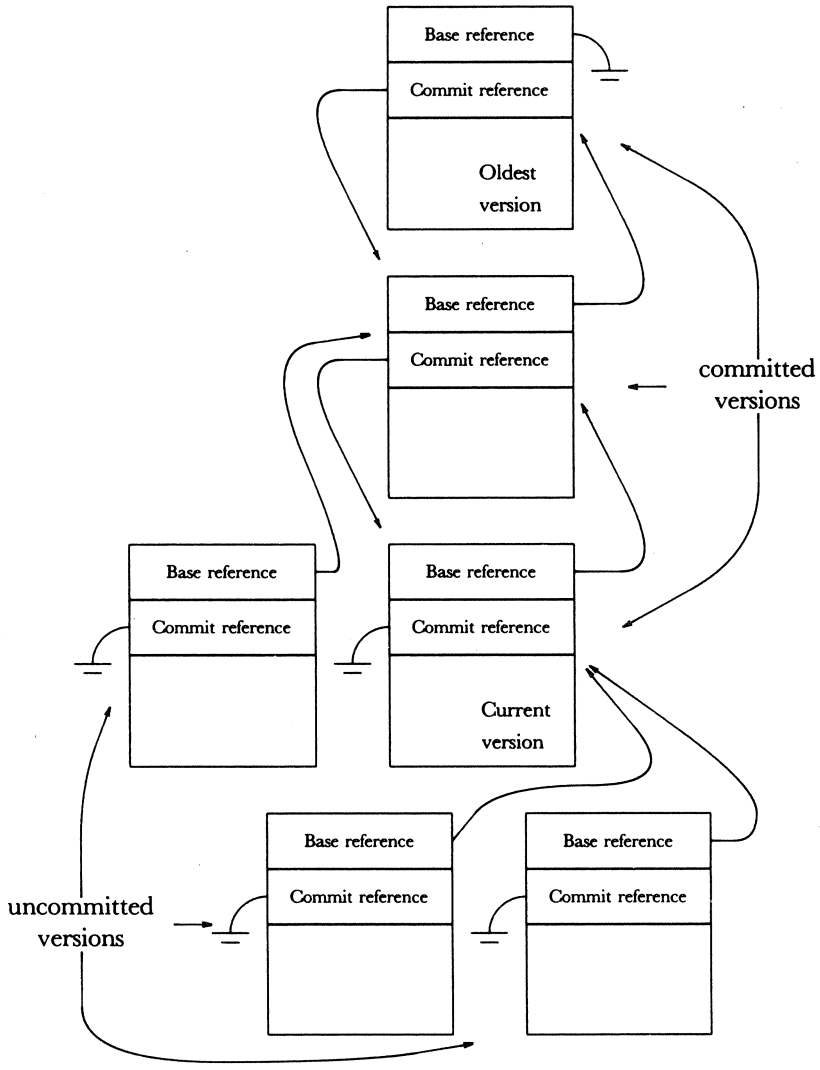


FIGURE 4. The *'family tree'* of a typical file. Only the version pages are shown. The page trees descending from the version pages are not shown.

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 also be read. This implies that, if a page has not been searched, then the subtree of which it is the root cannot have been searched 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 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 pagethe version pageis the only page that is written in place. When a page is thus copied, the *C* flag is set in the reference to it (in the parent page). Naturally, a page is 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 was written itself, and modified if a client explicitly requested the page tree to be changed, for instance, by adding or deleting pages.

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 a *C*, *R*, *W*, *S* and *M* flag 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.

5.2. The optimistic concurrency control mechanism

As long as updates are done one after the other, commit always succeeds and requires virtually no processing at all. When two updates are done concurrently, however, the server must check if commit can be allowed by testing if the two 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.

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 [Kung81]. The validation phase checks *serial equivalence* of transactions T_i and T_j by testing if 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 done in 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* commits 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 *.i* commits 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. This means that the read set of the version to-be-committed must be compared to the write set of the already-committed version. The already-committed version cannot change, so this test can be carried out without locking being needed, or critical sections. When the test succeeds, the version-to-be-committed is established as the successor of the already-committed version, and commit is attempted as if condition (1) holds.

When a client requests to commit a version that is based on the current version, condition obviously (1) holds, because it was created after the current version committed. Therefore, Amoeba File Service allows all commits of versions based on the current version. The mechanism for this is demonstrated in figure 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. *M.a* must then do the following without allowing other requests to interfere. First it must check if *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 done 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 block 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.

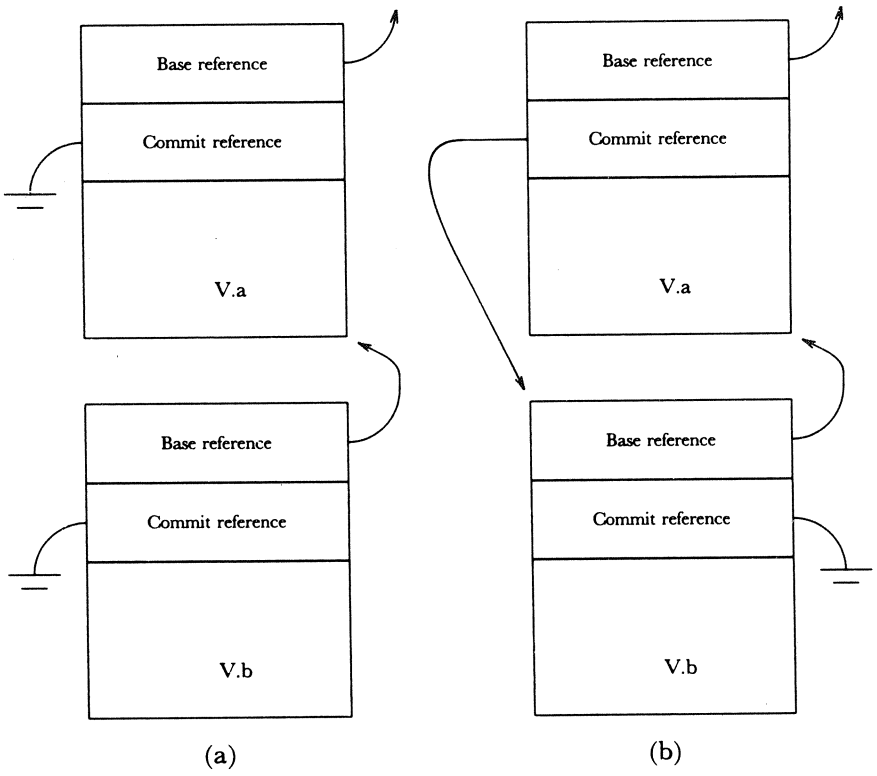


FIGURE 5. *V.b* succeeds *V.a* as the current version. (a) shows the situation before the commit, (b) shows the situation after the commit.

This is the only critical section in version commit: test and set the commit reference. In order to make this 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 5(a) shows the situation before commit, figure 5(b) after the commit has successfully been carried out. *M.b* returns an acknowledgement to *M.a* and *M.a*, in turn, returns an acknowledgement to *C*.

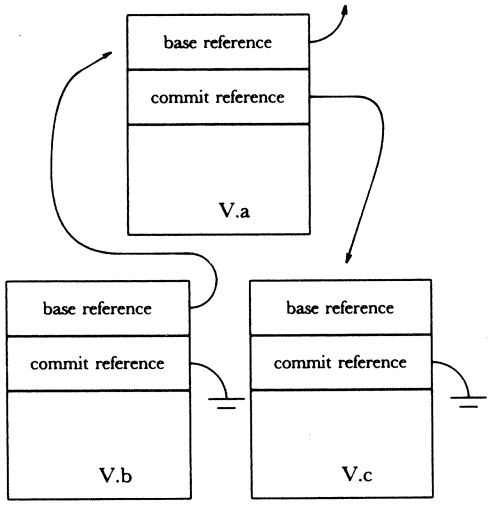


FIGURE 6. *V.b* wants to commit, but is no longer a descendant of the current version, *V.c*.

Let us now examine the case where *V.a* is no longer the current version, but another update, concurrent with that of *V.b*, has taken place. Let us assume the situation of ; *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 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 if 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 uncopied parts of the tree in either *V.b* or *V.c* need not be visited since they can neither have been read nor written.

While descending the two page trees, checking the serialisability constraint,

M.b also prepares the new current version, which must contain the updates made in *V.c* and 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.

5.3. The locking mechanism

In the previous section we have assumed the upper part of the file tree consists of only one version page. In this section we describe the mechanisms for updating files when the upper part of the tree consists of more than one version page.

Before continuing, some terms are defined to simplify discussions. The upper part of the tree, stored on magnetic media, 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 will be called a **small file**, although a 'small file' may, of course, be arbitrarily large. A file whose root is an internal node of the system tree will be called 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 greater 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 some drawbacks, however, especially with regard to crash

recovery. Most systems that use locking need elaborate mechanisms to restore the system after a crash: Locks have to be cleared, files or databases may have to be rolled back, or intentions lists must be carried out 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 are made of ports, which are 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.

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.

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* (with the same port, 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 unlocked page, or a page with a *top lock*. If the page thus found is not locked, the *inner lock* can be ignored. If the page is locked, it is treated as described above.

5.4. 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 race conditions thus introduced. The decreasing cost of primary memory makes caching techniques increasingly useful both for file servers and their clients. Some file servers have attempted a solution, the most prominent of which is probably XDFS [Sturgis80] Although XDFS provides an efficient mechanism for caching files or portions of files, the designers of the file server introduced the concept of the *unsolicited message*, a prod in the form of a message from server to client, telling the client his cache entry has become invalid. We have rejected such a solution because it does not fit the client-server model: an active client, that sends requests to a passive server that merely waits for requests, and carries them out. To have to be prepared to receive unsolicited messages makes client programs unnecessarily complex.

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 new version of a file is created, a client or a server examines its 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 server or a private client 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 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.

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. But our method of maintaining a cache is even more efficient for files that are not shared: the cache entry will always be for the most recent version of a file, so the serialisability test is a null operation, and all pages in the cache will always be valid.

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.

5.4.1. Robustness

The potential strength of distributed file systems, in contrast to traditional centralised file systems, is that distributed file systems can be much more 'crash proof'; that is, the file system will continue to operate, even when a few of the server processes, or even some of the disks are not operational.

Note that increased crash resistance and efficient concurrency control tend to mutually exclude each other, because better crash resistance is usually obtained by replication of data, which makes concurrency control more difficult. Making the Amoeba File Service crash proof has been an important aspect of its design.

In principle, the File Service operates using a number of server processes, which, in turn, use a number of block servers for information storage. This causes a separation of reliability aspects into two distinct areas: on the one hand, accessibility and robustness of file services as such, and, on the other hand, accessibility and robustness of individual files and versions. The former

is realised through replicated server processes; the latter through replicated block storage, such as, for instance, *stable storage*[Lampson79] and backup block servers.

Assuming stable storage is used, the pages of each version of each file that are on disk are, in principle, always accessible. Access paths to committed versions go through the replicated file table, and a chain of version pages on stable storage, hence version access and file access can be guaranteed as long as one or more servers are operational.

Uncommitted versions need not be salvaged in a server crash. The concurrency control mechanisms were designed such that clients must be prepared to redo the updates in a version; if a version is lost in a crash the situation is not much different. Uncommitted versions are therefore not as important as committed versions.

6. 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 optimistic concurrency control, 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, described in § 6.4.3, 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 super-files 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 applicationssuch as the example, mentioned in the introduction, of a compiler that needs to make temporary filesmust once again pay the price of all that complicated machinery for guaranteeing serialisability. This need not be the case at all: Pages of 32K bytes can be written. Often, one such page is 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.

References

[Dion80]
DION, J., "The Cambridge File Server," *Operating Syst. Rev.*, vol. 14, pp.41-49, Oct. 1980.

[Eswaran76]
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*, vol. 19, no. 11, pp.624-633, November 1976.

[Fridrich81]
FRIDRICH, M. and OLDER, W., "The Felix File Server," *Proc. Eighth Symp. Operating Syst. Prin.*, pp.37-44, 1981, ACM.

[Kung81]
KUNG, H. T. and ROBINSON, J. T., "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp.213-226, June 1981.

[Lampson79]
LAMPSON, B. W. and STURGIS, H., *Crash Recovery in a Distributed Storage System.* Palo Alto, CA.: Xerox PARC, 1979.

[Menasce]
MENASCE, D. and MUNTZ, R., "Locking and Deadlock Detection in Distributed Databases," *IEEE Trans. Softw. Eng.*, vol. SE-5, pp.195-202, May 1979.

[Mullender85]
MULLENDER, SAPE J. and TANENBAUM, ANDREW S., "Protection and Resource Control in Distributed Operating Systems", Report IR-79, Vrije Universiteit, Amsterdam, June 1985.

[Mullender86]

MULLENDER, S. J. and TANENBAUM, A. S., "The Design of a Capability-Based Distributed Operating System," *The Computer Journal*, vol. 29, no. 4, pp.289-300, 1986.

[Reed78]
REED, D., "Naming and Synchronization in a Decentralized Computer System," *PhD. Thesis*, 1978, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

[Reed81]
REED, D. and SVOBODOVA, L., "SWALLOW: A Distributed Data Storage System for a Local Network," *Proc. IFIP*, pp.355-373, 1981.

[Robinson82]
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.

[Rochkind75]
ROCHKIND, M.J., "The Source Code Control System," *IEEE Trans. on Softw. Eng.*, vol. SE-1, no. 4, pp.364-370, December 1975.

[Schlageter81]
SCHLAGETER, G., "Optimistic Methods for Concurrency Control in Distributed Database Systems," *Proc. VLDB Conference*, 1981.

[Stonebraker81]
STONEBRAKER, M., "Operating System Support for Database Management," *Comm. ACM*, vol. 24, no. 7, pp.412-418, July 1981.

[Sturgis80]
STURGIS, H., MITCHELL, J.G., and ISRAEL, J., "Issues in the Design and Use of a Distributed File System," *Operating System Review*, vol. 14, no. 3, pp.55-69, July 1980.

[Tanenbaum82]
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).

Immediate Files

Sape J. Mullender
Andrew S. Tanenbaum
Deparment of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

An efficient disk organisation is proposed. The basic idea is to store the first part of the file in the index (inode) block, instead of just storing pointers there. Empirical data is presented to show that this method offers better performance under certain circumstances than traditional methods.

1. INTRODUCTION

In many transaction-oriented computer systems, the performance bottleneck is accessing disk files. Consequently, when a file system is being designed, careful thought should be given to trying to minimize the number of disk accesses. In this note we discuss some measurements we have made on an actual file system, then we look at new kind of file organisation suggested by these measurements, and finally we compare the performance of the new file organisation to that of the UNIX† operating system.

While designing a free-standing transaction- (as opposed to connection-) oriented file server for the Amoeba [Mullender86] distributed operating system, we made some measurements of file sizes on our departemental UNIX system. The results are summarised in figure 1. For example, 60.87% of the 19978 files on the disk are 2048 bytes or less. The conclusion to be drawn from this data is simple: most files are short.

As an example of how file systems are typically organised, consider the UNIX file system [Thompson78]. Associated with each file is a 64-byte data structure called an *inode*. The inode contains some bookkeeping and accounting information plus 13 disk addresses. These disk addresses occupy three

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

Immediate Files
S. J. MULLENDER and A. S. TANENBAUM
Software-Practice and Experience
Vol. 14, No. 4, pp. 365-368
April 1984

File length	*Percent*	*File length*	*Percent*
1	1.79	1024	48.05
2	1.88	2048	60.87
4	2.01	4096	73.51
8	2.31	8192	84.97
16	3.32	16384	92.53
32	5.13	32768	97.21
64	8.71	65536	99.18
128	14.73	131072	99.84
256	23.09	262144	99.96
512	34.44	524288	100.00

FIGURE 1. Percentage of files smaller or equal to the indicated length.

bytes each. Each of the first 10 of these can point to a disk block containing some data. If disk blocks are n bytes long, files up to length $10n$ bytes can be accommodated in this way. For longer files, the eleventh address points to a disk block, called an *indirect block* containing $n/4$ disk block addresses. (A disk block address occupies four bytes.) For files larger than $(10+n/4)n$ bytes, the twelfth disk address in the inode points to a *double indirect* disk block that points to $n/4$ additional indirect blocks. Finally, for huge files, the thirteenth disk address in the inode points to a *triple indirect* block.

The inodes are located contiguously in a sequence of blocks near the start of the disk. When a file is referred to by its ASCII name, the directory system maps the string onto an inode number, which is then used as an index into the inode block to find the inode. Thus for files up to length $10n$ bytes, two disk access are required, one to fetch the inode and one to fetch the data block. (In a connection-oriented file system, the inode need only be fetched once, when the file is opened, but in a transaction-oriented file server that erases its tables after each request has been replied to, the inode must be refetched on each transaction.)

The combination of short files and the need to make two disk accesses suggests another possible file organisation: expand the inode to a full disk block, and put the first part of the file in it. We call this an *immediate block*, in analogy with an immediate operand to a machine instruction. If the block size is 2048 bytes, some 60% of the files can be accessed in only one disk operation. For larger files, access to parts of the file outside the immediate part, would require the same number of disk accesses (two, three, or four) as in UNIX.

Before we describe our file organisation in more detail let us compare the number of disk accesses required to read every byte in our sample of 19978 files for UNIX and for our proposed file organisation. We have also computed the storage efficiency using both immediate files and UNIX, again for the measured file length distribution. It is important to use actual length distributions because the whole concept of an immediate file only makes sense in light of empirical data showing that short files are common. The results of these calculations are given in Fig. 2. The two columns labeled "Percent disk

storage wasted" were computed by $(A-L)/A$, where A is the amount of space allocated to files and inodes, and L is the total length of the files (i.e., the user data).

	Disk accesses per read/write		Percent disk storage wasted	
Block size	*UNIX*	*Immediate files*	*UNIX*	*Immediate files*
512	2.12	1.69	8.3	13.9
1024	2.06	1.46	13.3	14.4
2048	2.02	1.29	22.2	22.0
4096	2.01	1.16	36.7	36.6

FIGURE 2. Disk accesses and storage efficiency for various block sizes.

The conclusion to be drawn from this study is that immediate files can provide improved response times for transaction-oriented file servers. If the block size is small, the response time is improved at the cost of less efficient use of storage, but when the block size becomes 2048 bytes or more, immediate files are a little less wasteful than UNIX files. Furthermore, we conclude that the relative advantage of immediate files over the UNIX organisation increases with increasing block size.

We shall now describe the organisation of immediate files in more detail. When a file is created, an inode block is allocated. Unlike UNIX, inodes need not reside at the beginning of the disk, but may be located anywhere. The last 48 bytes of an inode block are reserved for the inode. The rest of the block is used for immediate data. The structure of the inode can be exactly as in UNIX, with the exception, that only 24 bytes are available for block pointers, whereas UNIX has 40 bytes worth of pointer space. These 24 bytes are used for 5 pointers to direct blocks, and one pointer each to an indirect block, a double indirect block, and a triple indirect block, giving 8 pointers altogether. Each pointer block can contain $(n\text{-}48)/4$ pointers, and a data block can contain n-48 data bytes, since the last 48 bytes of each block (the inode space) remain unused for pointers or data. This space may be used to hold recovery information for possible disk crashes and *hints* to make sequential file access even faster [Lampson79]. The file organisation is illustrated in figure 3.

If we assume a block size of 2048, then 2000 bytes are available in every block for pointers or data. Of every file, the first 2000 bytes are in the inode block, the next 10,000 bytes are in five direct blocks, pointed to by the five direct pointers in the inode. The next 1,000,000 bytes are in 500 indirect blocks, pointed to by 500 pointers in a pointer block, pointed to by the indirect pointer in the inode. There can be $500\times500\times2000$ or half a billion double indirect bytes, and $500\times500\times500\times2000$ or 250 billion triple indirect bytes. Since most disk drives are less than 500 Megabytes, it is also possible, when the blocksize is 2048 bytes or more, to use inodes with six direct blocks, one indirect block, one double indirect block, and *no* triple indirect block, since it is not needed then.

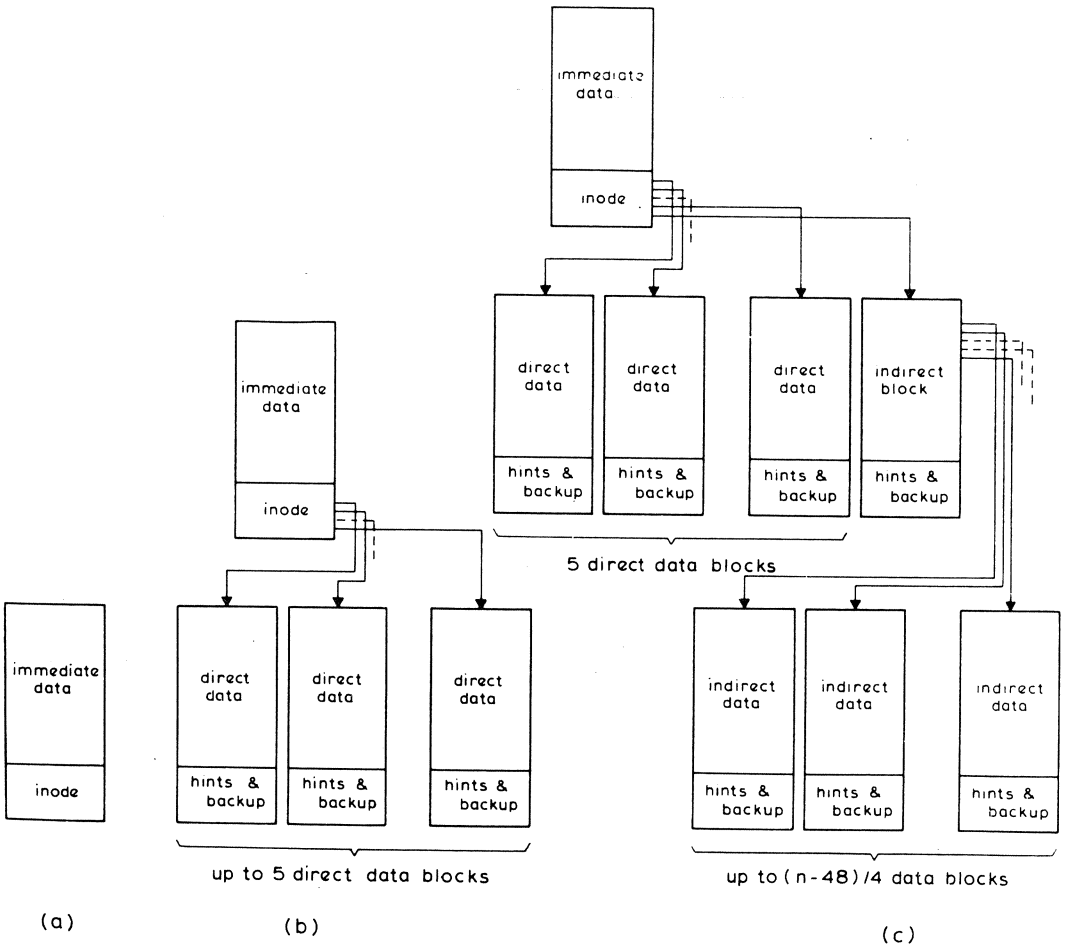


FIGURE 3. (a) immediate file, (b) direct file, (c) indirect file.

ACKNOWLEDGEMENTS

We would like to thank Dick Biekart and Bram Janssen for valuable discussions.

REFERENCES

[Lampson79]
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.

[Mullender86]
MULLENDER, S. J. and TANENBAUM, A. S., "The Design of a

Capability-Based Distributed Operating System," *The Computer Journal*, vol. 29, no. 4, pp.289-300, 1986.

[Thompson78]
THOMPSON, K., "UNIX Implementation," *Bell System Tech. Journal*, vol. 57, pp.1931-1946, July-Aug. 1978.

Wide-Area Networks

Distributed Systems Management in Wide Area Networks

Sape J. Mullender
Centre for Mathematics and Computer Science
Amsterdam, The Netherlands

While quite a few distributed operating systems for local-area networks exist, hardly any work has been done to date on distributed operating systems for wide-area networks.

In Europe, a number of public networks are now operational, with gateways between some of them. However, the use of these networks is still mostly restricted to *"remote login"* and, in some cases, simple *file transfer* operations.

To study these problems and to find structural solutions for efficient and simple use of national and international networks the working group *"Distributed Systems Management"* was founded within COST 11. Recently, this working group has submitted a research proposal to COST 11 to realise an infrastructure for the implementation of distributed services in a wide-area network in a European collaborative effort. The model, underlying the reserach is the *service model*, used in many local-area network distributed operating systems.

The research project is described, and the proposed infrastructure is discussed in some detail.

1980 Mathematics Subject Classification: 68A05, 68B20.
CR Categories: C.2.2, C.2.4, D.4.4, D.4.7.
Keywords & Phrases: service model, distributed systems, wide-area networks, COST-11

1. INTRODUCTION

Many distributed operating systems exist, based on local area networks, but, in spite of a growing need, the possibilities to use the potential of national and international networks efficiently are virtually non-existent.

Some networks are now operational in Europe, with gateways connecting them here and there. In principle information could be exchanged on these networks. Currently, however, these networks are almost solely used for *remote login, electronic mail, teleconferencing* and *file transfer*. Most of these

Distributed Systems Management in Wide-Area Networks
S. J. MULLENDER
Proc. NGI/SION Symposium
Amsterdam
pp. 415-424
April 1984

applications have been developed on an *ad hoc* basis, each application with its own protection mechanisms, network protocols, etc. Using an international network for any other applications often requires nested log-in on a number of hosts on the path through various networks to the destination host.

The COST 11 [Martin-Löf83, Kalin83] *"Distributed Systems Management"* group has been started to study these problems and find an infrastructure for simplifying management of distributed processing activities. In january of this year, the COST 11 DSM group finished a research proposal for a four year programme of collaborative research on some of the issues of distributed system's management in Wide Area Networks. The work will be carried out by research institutes all over Europe. In The Netherlands, participants are the *Centrum voor Wiskunde en Informatica*, the Computer Science Department of the *Vrije Universiteit* and the Network Group of the *Technische Hogeschool Twente.* COST 11 is asked to finance the collaboration costs, such as travel and subsistence cost, network connections and communication costs. This paper discusses the proposed research.

2. REQUIREMENTS, PROBLEMS AND ISSUES

The principles underlying the management of information processing systems apply whether the systems are local or distributed. In the present context a three part definition of management is used:

i. management is planning and organising the provisions of resources and identifies (a) where resources may be located, (b) their availability and (c) their cost;
ii. management is the control of the use of, and access to, resources according to allocation, optimisation and authorisation rules;
iii. management is the task of ensuring that resources remain accessible and that they function correctly; and, when this objective cannot be achieved, of ensuring that suitable signals are available which identify the failure.

This definition is wide ranging, covering management both within and external to individual network hosts. To narrow down the area addressed by distributed systems management it is important to differentiate between local management activities of the various host operating systems within the network and those activities concerned with the distributed activities of the system. The open systems' environment offers a set of services provided by the host operating system. The way in which those services may be implemented is outside the scope of open systems interconnection. Standards for Distributed Systems Management are concerned with a non-local use of these services. However, the interaction between local and distributed aspects of management are a significant R & D matter.

Management is realised through the actions of managers. It is important that the managers of host systems (i.e., people) have the freedom to effect the management policies appropriate to their systems. Distributed systems management must provide the framework for general mechanisms in which a variety of management policies can operate.

Given the functions of management and the understanding of the constraints imposed by wide area networks, the tasks to be addressed are:

1. to identify models for distributed systems as a context within which management activities can be considered;
2. to identify (and, if possible, develop) the set of management tools which assist in the planning and organisation of distributed processing;
3. to identify the mechanisms through which managers can apply their policies to control and use resources, through appropriate optimisation and authorisation strategies;
4. to identify protocols for the control of resources, protocols which maintain resource availability and protocols which signal system failure.

For network users and managers to have their requirements satisfied a number of mechanisms and services need to be provided. Perhaps the greatest barrier to offering such mechanisms and services in a Wide Area Network is the lack of an agreed model for the organisation of distributed computing and for a set of communication standards for the exchange of control and supervisory information.

Even with such a model and a set of management communication protocols there are still detailed problems to be resolved concerning the details of the management mechanisms and services which are needed. The more important ones concern mechanisms for service location and authorisation, and the lack of quantitative information which can offer guidance to management in controlling the resources for which it is responsible. Of lesser, but still significant, importance are the services which assist users to obtain the most effective use of the facilities offered by the network of distributed computers.

At present, there do not appear to be either the data to help resolve these issues nor any general models of distributed systems through which these issues can be investigated. Managers need realistic data from efficiently managed distributed systems to feed into their models in order to help them with their planning. Yet, at the same time, managers are unable to establish whether their systems are operating efficiently for lack of adequate diagnostics and tools to help them analyse distributed system performance.

Managers are unlikely to accept other than the most stringent safeguards in the application of authorisation rules. The global access which is (theoretically) possible in a Wide Area Network and the fact that management units have autonomy means that users and the system they use have to carry out an authentication excercise at the start of any instance of a communication session. Subsequent dialogue must be policed by the computers within the Wide Area Network (even by the Wide Area Network itself) to maintain the integrity of the session. The most effective authentication mechanism and the way that mechanism is made apparent to both users and computer or network manager still requires detailed study.

3. The service model for distributed processing

Today most people use computers interactively; that is, they type commands at their terminals, the system will process their commands and return a reply. If the user is satisfied with the answer, he may type a new command; if not, the user may retry the command, or phrase it differently. Inside a computer's operating system the same thing goes on, albeit at another level: the user's command interpreter makes system calls, requesting programs to be run, files to be read, tapes to be rewound, etc., and the system replies with data, or simply with an acknowledgement. At a lower level still, programs make extensive use of subroutine calls; the call can be seen as a request to execute the body of the subroutine, and the subroutine return as its reply. Obviously, thinking in terms of requests and replies, possibly nested recursively, is an excellent way of structuring problems into small portions.

Conceptually, distributed systems are among the most difficult to oversee, so a structured approach to building distributed systems is essential. The natural choice of a model is that of using requests and replies. In this section we shall examine this model in some detail and discuss its consequences on distributed systems design.

The maker of a request shall be named the **client**, the processor of the request shall be the **service**. A client can be a person at a terminal, an operating system, a process, a processor, etc. A service is an abstraction of the requests that can be made and the replies that can be expected, comparable to *abstract data types* [Liskov74]. A service is always embodied by one or more **servers**, the processes, processors, or devices that carry out the requests as defined by the service.

A **request** consists of information travelling from client to server, a **reply** is information sent by the server back to the client. In a distributed system client and server do not generally reside on the same physical machine; requests and replies must therefore be sent through a computer network from one host to another. Depending on the type and speed of the network, requests and replies can be sent as packets, messages, or over connections.

So far, the service model closely resembles a remote procedure call mechanism, the request representing the call, and the reply the return. It is more than that, however: unlike a subroutine, a service can fail. The processor where the service is implemented may stop working, a bug in the service program may cause the server to crash obscurely once every thirteen weeks, or the network may break down. Making a request is like calling a subroutine that *almost always* returns. In a program, a non-returning subroutine causes the program to fail; in a distributed system, a non-replying service need not crash the client. The client can retry a number of times, expecting the service to be repaired, or to contact another instance of the service, or it can resort to another service to achieve its goals in a different way.

The property of distributed systems of potentially surviving server crashes is what makes distributed systems so interesting from the point of reliability and error recovery. But it is necessary to design the software to expect errors, and to react to them appropriately. A client must *expect* a server to crash every

now and then. When a reply does arrive the client may always assume the server has done its work correctly, but if no reply comes, the client does not know if its request has been carried out; the client must try to find out, and, if necessary, repeat the request.

In the same philosophy, services must be designed in such a way that recovery from crashes can be simple and straightforward. Most request can be so defined that carrying them out once, or more than once does not yield different results. If a server crashes, such requests can safely be repeated.

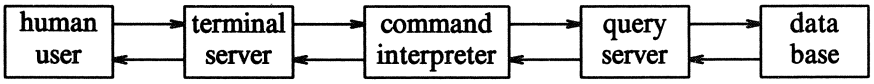


FIGURE 1. An example of a service hierarchy

Naturally, a server can itself be client to another service. In fact, the possible hierarchy of services is the strength of the model. As an example, a possible hierarchy of services is shown in Fig. , where a human being is shown as a client of a terminal server, which in turn, is a client of the command interpreter, etc. A crash, for instance, of the database server, will be detected by the query server, which must then try to recover from it. The query server can retry the request, it might rephrase a query to get the answer from another database server, and as a last resort, it can report failure to its client, the command interpreter. In this way the human client at the top of the hierarchy gets to cope only with irrecoverable errors and crashes in the system.

4. SERVICES IN TRADITIONAL OPERATING SYSTEMS

Traditional operating systems provide service to its clients in a much more restricted way than conceived in the service model of the previous section. Usually, the only services available to programs are those provided by the operating system in the form of system calls. This restricts the service hierarchy to two levels: user to program, and program to operating system. Some operating systems, such as UNIX,† have a well designed user-program interface: to the client-user a number of alternative services is available, and programs can sometimes be combined to provide powerful "programs of programs," but even among the best of operating systems, the possibilities are limited, and, at the system call level, no choice of service is left to the programmer at all. Most operating systems, for instance, have a built-in file system, and, whether one likes it or not, it is the only available file system.

Traditional operating systems run on one centralised processor; if it, the operating system, or one of its components (file system, terminal controller, etc.) crashes, the whole system crashes. Naturally, in these systems it is not necessary to pay much attention to recovery from crashes: if the system crashes, nothing can be done anyway. Sometimes, operating systems run on

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

more than one processor, but even then, the processors are so closely coupled that a crash in one brings down the others also. If we want to use existing systems as a basis for reliable crash-proof distributed systems, we must add mechanisms for error recovery, and increase the possibilities of allowing clients the choice of many services.

Many computer centres now have a connection to one of the wide area networks, so communication is possible between one computer and another. The services available over the network are very limited, however. Often there is a network-wide electronic mail service, and sometimes there are file transfer capabilities. Occasionally we find another special-purpose application that can be accessed through the network, such as teleconferencing systems, but only in very few cases does the operating system allow processes on one host under one operating system to communicate with another process in another host under another operating system.

5. Integrating the service model with existing software

In the short term, computer networks will be mainly used for mail exchange, file exchange, and remote file and data base access. In a primitive form this is already provided on many existing systems. Accessing these services requires intimate knowledge, however, of both the system where requests originate and the system where the service is implemented.

In the long term people will have to use the network more intensively and for many more types of access. If computer networks still work in the same fashion, the expert knowledge required to use these services will increase dramatically. It is therefore essential that a uniform way of accessing remote services is inserted between the operating system and the (remote) client. This model is shown in figure 2.

An essential property of this model is that it allows existing software to be integrated into a distributed environment. If new software is henceforth written directly in the language of clients and servers, requests and replies, reliability and error recovery, a gradual changeover to practical distributed systems on a large scale is possible. Several tries have been made in the past to build a coherent distributed system on top of existing operating systems [Thomas73, Millstein77, Mamrak82]. We also mention [Hall80] which is an attempt to build a uniform user interface on a collection of operating systems, an approach very similar to the uniform client-server model.

6. The service model as the operating system of the future

As processes shall rely on fewer services of the local operating system, but rather on services replacing traditional file i/o, terminal management, etc., the underlying operating systems will become increasingly simple. This is fortunate, because today operating systems are among the most complicated programs written. Few, if any, operating systems are free of bugs, and we believe the main cause lies in their complexity.

We must set as our goal to reduce the complexity of the operating systems, by removing every function from it that can also be realised outside the

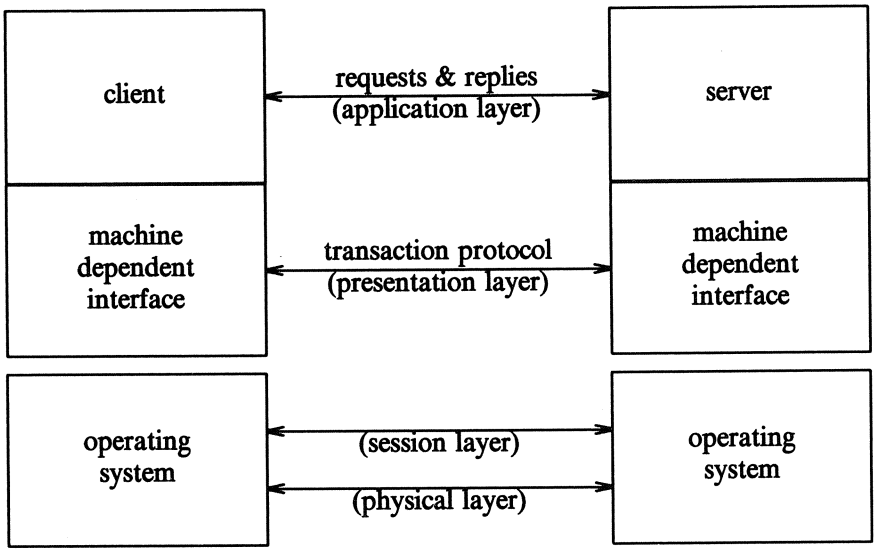


FIGURE 2. The client server implementation model for existing computer systems. The double line represents the separation between application software and the operating system.

operating system. Eventually, the only task the operating system has, is to provide programs with an environment for execution, and interprocess communication primitives. Remaining functions will then be memory management, exception handling, and the implementation of system calls for interprocess communication and process control (allocate memory, ignore or catch certain exceptions, timers, exit, etc.). It is likely even that processors will become so cheap that it is no longer worth while to implement multi-programming, but assign one process per host. From the viewpoint of protection and scheduling this can be a great simplification of the operating system.

Process creation can be done through the services of the **process server**, a service that finds a suitable processor for the process to be run, downloads the code into it and starts the processor. Accounting can be done by an accounting service, the **Bank Server**, to be described later. Different file systems can co-exist to give the users maximum choice of service, interface, reliability and speed. The nature of the storage system for a database is completely different from the one needed by, for instance, a compiler that makes a temporary file, and different again from the storage needed by a text editor. This is indeed why today database systems often run on separate machines; the file system provided by the "regular" operating system is unsuitable for database applications [Stonebraker81, Tanenbaum82].

Existing software can be ported to the new generation distributed operating system by building an emulation layer that translates archaic system calls into

the new service requests.

7. Management of services in wide area networks

An important difference between distributed systems in local area networks and those in wide area networks is that local systems are usually under control of one administration, while wide area networks are usually under control of many different administration. In wide area networks the lower layer communication protocols are usually provided by the PTTs so the choice of using datagram service or virtual circuits is not available.

Each administration in a wide area network potentially has its own procedures for accounting, resource control, and access permissions. For wide area distributed systems, it is important that one accounting and resource control mechanism is available that can be used to realise all the different policies.

Basically, there are two methods of access control. One is to use *access control lists* (ACLs), the other is to use *capabilities*. Both methods are well known: In the ACL method, a server grants a client access to an object after checking if the client is on the object's list of authorised users. In the capability method, a server grants a client access to an object if the client can present a capability for the object. The first model is characterised by a list of authorised clients, stored with each object; the second by a list of capabilities for objects to which access is allowed, stored by each client.

The ACL method requires a way for a server to establish the identity of its clients. It may not be possible that a client impersonates another to obtain access to an object (or service) that would have been refused otherwise. The capability method requires a method of distributing capabilities to clients in such a way that clients cannot forge them, construct them, or obtain access to an object by trail and error. For both problems adequate solutions exist [Mullender84, Donnelley80, Evans74, Needham78]. Both methods should be supported by the interprocess communication mechanisms to allow different administrations to use different access control policies.

8. Management supporting services

A number of services are conceived to support usage and management of Wide Area Network services. Among these are *Name Service* which map local, private names for objects onto globally unique object names, *error reporting* to help managers detect malfunctioning network components, *performance monitoring* for managers to detect bottlenecks in the system, *Bank Service* for accounting and resource control, *help service* for the assistance of users who get stuck, and *command interpreters* to help users to communicate with Wide Area Network services in a natural and meaningful way. We shall discuss some of these services below.

Name Servers or **Directory Servers** provide a mapping between clients' private name spaces and globally unique object or service names. A further mapping maps global object and service names onto the network address of the object or service. This two-level mapping allows a clean separation of functionality: when a client renames an object, only the first mapping is

affected; if an object migrates to another host, only the second mapping is affected.

Directory Service thus consists of two more-or-less independent services: a service in the user domain, for conveniently naming private objects, and a service in the operating system domain, for locating objects, given their globally unique name. This separation allows the existence of several independent directory services in the user domain, offering different capabilities. Directory services could offer *"yellow pages service"* which responds to queries of the nature: "Tell me the names and give me a description of file servers that implement atomic update and concurrency control mechanisms."

The global-name to network-address mapping is the subject of considerable research. This map has to be carried out efficiently, and it has to be carried out securely. It is obviously unacceptable if requests, containing sensitive information for a particular trusted service, end up on the wrong host. Service location and object location is closely related to issues of authentication, protection, and encryption, and the DSM group intends to investigate the problem in this context.

An example of a versatile accounting mechanism that can be used for resource control, access control, and, of course, accounting is the **Bank Server**, described below.

The **Bank Service** consists of one or more Bank Server processes that maintain *accounts* for each user in the system. An account may contain *"virtual money"* in one or more *"virtual currencies."* One of the currencies could correspond to *"real money"*. Other currencies can represent disk quota, cpu seconds, phototypesetter pages, etc. A service can ask the Bank Service to make a new currency for it, specify the amount of money to be *coined*, and hand out the money to its (potential) clients, possibly in return for virtual or real money in another virtual or real currency.

To make the Bank Server secure, it uses a capability mechanism; the user that creates an account receives a capability for it. Only by presenting the capability a client can take money out of an account. Keeping the capability of an account secret is the key to preventing other users from stealing one's money.

A typical interaction between a client and a server goes something like this: first the client, presenting a capability for his account, requests the bank service to prepare a cheque for some amount; the Bank Service debits the client account, makes a unique unforgeable bit pattern representing the amount, and returns that to the client as a cheque for the amount. The client then sends his request to the server along with the cheque, and the server clears the cheque with the Bank Service before carrying out the request; the Bank Service credits the server account with the amount, and erases the bit pattern in the cheque from its list of outstanding cheques, preventing a cheque from being cashed twice.

Doing business like this requires two extra transaction with the Bank Service for every transaction that has to be carried out, but, fortunately, it can easily be optimised. The client can send an amount to the server that covers many

transactions in one blow, the server can cash the cheque once, and maintain a local credit account for each client for which it works. The amount sent at one time by a client to a server must not exceed the amount of trust the client has in the server.

A mechanism as just described can be made very secure. A property that could make the accounting system desirable in an international environment is that untraceable payments can easily be implemented [Chaum82]. The Bank Service is not in a position to analyse a user's spending patterns in this way.

Network users need meaningful messages when interacting with the services provided by the system. The purpose is to make the users' interaction with the network and its facilities as effective as possible. Users are unlikely to use the range of facilities which a network can offer unless a user-friendly environment is available. Observation that a network's facilities are easy to use and that *"Help"* is readily available will act as a catalyst to promote others to use them.

A **Help Service** can operate in four ways:

i. giving users assistance when there are faults, e.g., how long it will take before a service is resumed, or whether the fault led to any loss of information;
ii. giving guidance on how to access services, e.g., by providing on-line documentation, structured walk-throughs for novices, and, in the last resort, human contact points for further information;
iii. offering a focal point for user feedback, e.g., customer complaints and requests;
iv. providing users with status information, e.g., on service or network availability, maintenance schedules, and advising on (advertising) new services.

9. Programme for research and development

Having considered the evidence for distributed systems management and presented lists of options for tools and services, it is now appropriate to draw some conclusions.

One of the motivating forces for the Distributed Systems Management study carried out by the COST 11 bis DSM-Group was the realisation that standardisation bodies were having to grapple with issues which are still lively research topics. The analysis presented in the report show that they remain research topics. Although some studies have been carried out and some systems have been built to demonstrate principles, they have not been withing the scope of open systems interconnection nor of the open use of wide area networks. Therefore, if standardisation work is to receive any support from this type of activity, it has to come from relevant practical exploration of the issues and of the proposed methods for carrying out distributed systems management. A further reason why standardisation bodies are having difficulties in this area is the interdisciplinary nature of the problem. Although distributed information processing has been facilitated by the development of adequate data communications, many of the key issues relate to standards for the user interface (OSCRL) and to matters which cannot be the subject of standards such as the

services and tools that are provided to support management.

The research programme consists of two related activities.

1. an investigation of tools which can enable managers to perform their functions more effectively;
2. an investigation of the services which are needed to provide a distributed system management infrastructure.

This work will need to research the types of model which should be developed and the general applicability of those models. At present, it appears that, whereas a single, more general model would be a desirable commodity in the long run, in the short term there is not yet the information available upon which to build more general models. A more pragmatic approach to model building to analyse particular and well identified scenarios is therefore recommended.

In meeting the requirements of the second activity a list of topics for study, development and implementation can be drawn up. The following list proposes a priority order:

1) The distributed system management kernel.
2) Name Servers
3) Authentication mechanisms
4) Journalling and performance monitoring
5) Help Services
6) Error reporting and diagnostics
7) Bank Service (accounting)
8) Command interpreters

It is noted that the technical solution to providing many of these management services is by the simple expedient of passing messagesin well specified formatfor storage within or retrieval from an information base. In the short term the message facilities provided by Computer Based Message Services (e.g., GILT [Wallerath83] perhaps) could suffice. The accounting requirements are not dissimilar from those put forward in the proposal to COST 11 for the OSIS project. Also the provision of "Help," the concern for adequate user interfaces and the legal implications of distributed information processing are of significance to those concerned with Human Factors [Eason83]. Thus, the study of Distributed Systems Management has synergystic relevance to other

COST 11 activities.

REFERENCES

[Chaum82]
CHAUM, D., "Blind Signatures for Untraceable Payments," *Proc. Crypto*, 1982, Plenum Publishing Co. N.Y..

[Donnelley80]
DONNELLEY, J. E. and FLETCHER, J. G., "Resource Access Control in a Network Operating System," *ACM Pacific '80 Conf.*, Nov. 1980.

[Eason83]
EASON, K.D. and JENSEN, W., "Human Factors in Teleinformatics," *Proc. of the European Teleinformatics Conf.*, pp.3-5, October 1983.

[Evans74]
EVANS, A., KANTROWITZ, W., and WEISS, E., "A User Authentication Scheme Not Requiring Secrecy in the Computer," *Comm. ACM*, vol. 17, no. 8, pp.437-442, August 1974.

[Hall80]
HALL, D. E., SCHERRER, D. K., and SVENTEK, J. S., "A Virtual Operating System," *Comm. ACM*, vol. 23, no. 9, pp.495-502, Sept. 1980.

[Kalin83]
KALIN, T., "Technical and Organisational Overview of COST 11 Bis Projects and Working Groups," *Proc. of the European Teleinformatics Conf.*, pp.3-5, October 1983.

[Liskov74]
LISKOV, B. and ZILLES, S., "Programming with Abstract Data Types," *SIGPLAN Notices*, vol. 9, pp.50-59, April 1974.

[Mamrak82]
MAMRAK, S. A., MAURATH, P., GOMEZ, J., JANARDAN, S., and NICHOLAS, C., "Guest Layering Distributed Processing Support on Local Operating Systems," *Proc. 3rd Int. Conf. on Distr. Comp. Syst.*, october 1982.

[Martin-Löf83]
MARTIN-LÖF, J., "The COST Framework and its Activities in Teleinformatics," *Proc. of the European Teleinformatics Conf.*, pp.3-5, October 1983.

[Millstein77]
MILLSTEIN, R. E., "The National Software Works: A Distributed Processing System," *Proc. ACM Annual Conf.*, pp.44-52, 1977.

[Mullender84]
MULLENDER, S. J. and TANENBAUM, A. S., "Protection and Resource Control in Distributed Operating Systems," *Computer Networks*, vol. 8, no. 5,6, pp.421-432, 1984.

[Needham78]
NEEDHAM, R. M. and SCHROEDER, M. D., "Using Encryption for

Authentication in Large Networks of Computers," *Comm. ACM*, vol. 21, no. 12, pp.993-999, December 1978.

[Stonebraker81]
STONEBRAKER, M., "Operating System Support for Database Management," *Comm. ACM*, vol. 24, no. 7, pp.412-418, July 1981.

[Tanenbaum82]
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).

[Thomas73]
THOMAS, R. H., "A Resource Sharing Executive for the ARPANET," *Proc. NCC*, 1973.

[Wallerath83]
WALLERATH, P., "The GILT Abstract Model of a Computer Based Message System," *Proc. of the European Teleinformatics Conf.*, pp.3-5, October 1983.

Connecting RPC-Based Distributed Systems Using Wide-Area Networks*

Robbert van Renesse
Andrew S. Tanenbaum
Hans van Staveren
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Jane Hall
Computer Science Division
Hatfield Polytechnic
Hatfield, England

Remote Procedure Call (RPC) is a widely used communication mechanism in local network based distributed operating systems. It is simple, fast, and straightforward to implement. However, when two or more distant distributed systems are connected, problems arise concerning the protocols, locating services, and other issues. To solve these problems, gateways are introduced. In this paper we discuss various ways in which these gateways can be organized and show how their application in the Amoeba Distributed Operating System has solved the problems cited above.

1. Introduction

As networks of high-performance personal workstations become more widespread, interest in distributed operating systems to make the whole system look like a single time-sharing system is increasing. When the same distributed operating system is running on two widely separated local-area networks, it is natural to think about merging them into a single transparent distributed system. However, because local-area and wide-area networks have very different properties, a number of problems arise. These problems and some proposed solutions are the subject of this paper.

* This research was supported in part by the Netherlands Foundation for the Advancement of Pure Research (Z.W.O.) under grant 125-30-10.

Connecting RPC-Based Distributed Systems Using Wide-Area Networks
R. van Renesse, A. S. Tanenbaum, J. M. van Staveren, and J. Hall
Informatica Report IR-118
Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam
December 1986

A brief outline of this paper follows. First, distributed operating systems are discussed, in particular those properties which make them hard to extend to wide-area systems. Next gateways are discussed. Several types of gateways are distinguished using the ISO OSI reference model [Zimmermann80]. Then attempts at transparent wide-area extension are described for the Cambridge Distributed Computing System, the V-System, and the Amoeba Distributed Operating System. Finally the solutions are compared.

This work was undertaken as part of the COST-11 ter MANDIS project partially sponsored by the European Community. In this project, two Amoeba Distributed Operating Systems in Holland (Vrije Universiteit, CWI), two in the U.K. (Harwell, Hatfield Polytechnic), and one in Norway (University of Tromsø) are being connected into a single, transparent distributed system as a research project, with the aim of investigating the tools and services required when interconnecting autonomous management regimes.

2. Distributed operating systems

When computer networks first appeared, the operating systems used on the computers were just ordinary operating systems extended with networking primitives. Using these primitives it was possible for a process on one computer to set up a connection or *virtual circuit* to a process on a remote computer. This communication was not transparent because the syntax and semantics of intramachine communication were different from intermachine communication. Such a system is called a *network operating system*.

The next evolutionary step in this direction was to try to hide the machine boundaries, so that to the user, the collection of machines would look and act like a single multi-user time-sharing system instead of a collection of autonomous machines. This led to the concept of a *distributed operating system*[Tanenbaum85] in which all the machines ran the same operating system kernel and handled resource management automatically. For example, in a distributed system, when a process or file is created, it is up to the operating system, not the user, to decide where to place it.

These differences in approach also led to differences in protocols. Network operating systems tend to do infrequent bulk file transfers, which can best be handled by connection-oriented sliding window protocols. On distributed operating systems, processes tend to have frequent short interactions with other processes, leading to connectionless *remote procedure call*[Birrell84] as the most widely used communication model.

Wide-area networks, like network operating systems, generally use connection-oriented protocols such as the ISO OSI protocols. When two RPC-oriented distributed systems need to communicate over a wide-area system, problems arise due to the different communication styles.

Another important property of distributed systems is how they locate processes and services. If a client, C, calls a server, S, the system must have some way of locating S. One approach is to have a central name server that maps S onto the machine number where S is located. The other approach is to broadcast a message asking all machines on the network if they know where S

is. Neither of these approaches is suitable when extending a distributed system to multiple remote sites. Something else is needed.

3. GATEWAYS

Whenever two networks are connected, a gateway machine is needed between them [Sunshine77]. The gateway has to deal with problems caused by

- different name spaces
- different packet sizes
- different protocols
- support of broadcast

In addition, gateways can play a role in flow control, congestion control, service location, and protection. In this section we will see how gateways deal with these problems, and on which level of the ISO OSI reference model they are taken care of. We shall see that it is not possible to put all gateway functionality below the transport layer, as proposed by OSI. An advantage of putting the gateway as low as possible in the layer hierarchy is that higher-level software does not have to differentiate between different networks (since there is only one logical network). However, we will show that it is not necessary for higher-level software to be aware of different networks even if gateways are at the highest layer.

On the lowest level in the OSI hierarchy is the *connector*, a simple, theoretical, gateway that physically links up the underlying cables of the network. Somewhat higher in the physical layer is the *repeater*, which amplifies the signal before transfering it to "the other side." *Selective repeaters* filter out messages that are not intended for the other side. They are in the data-link layer. Both connectors and repeaters have the property that they are completely software transparent, even where timing is concerned.

On the data-link level we find *relays*, which receive complete packets and transfer them to the intended networks. Relays are also software transparent when timing is not critical. High in the data-link layer we find relays that connect physically different networks, and make them look like a single physical network. The packet size on this network is the minimum of the packet sizes of all the cooperating networks, since the relay does not do fragmentation. Addresses of the new logical network are mapped to physical network and site address using routing tables. Usually, however, this is done on higher levels.

Network-layer gateways support different networks, and they fragment packets if they are too large for the destination network. Network-layer gateways can also help to avoid congestion in the network by re-routing packets.

Transport-level gateways understand transport protocols, and can "adjust" them for different networks, for example, by filtering out retransmissions that arrive too fast for the destination network. Moreover, they can do address translation: the local address is mapped to a remote address. Local addressing can therefore be independent of network-wide addressing. These gateways are implemented by having the gateway "stand in" as the remote process, that is, the local process thinks that it is talking to another local process. In reality,

the other local process is a half-gateway that transfers messages to and from another half-gateway on the remote network.

Session-level gateways work at the session layer and above. Here they have full control over participating networks, simplifying management and enhancing flexibility. Communication over these gateways is efficient, since the participating networks can be optimized for their local characteristics (effectively they do message reassembly). This is especially true if the networks differ considerably in bandwidth, and there is not much to be gained in forwarding packets before all have arrived. Flow control problems are taken care of on either side of the gateway.

3. Implementations

Many local-area networks are currently in use. However, only a few of them have been connected transparently over a wide-area network. The advantages of connecting these local systems are obvious—connecting them transparently allows applications to work unchanged across local network boundaries. In this section we will have a look at some of these systems, and how they have been implemented.

3.1. The Cambridge distributed computing system

The Cambridge Distributed Computing System [Needham82] is an experiment of the University of Cambridge to provide a computing system consisting of processors connected by a fast communication network, the *Cambridge Ring*. Some of the system's processors perform dedicated services, such as a name service or a file service, whereas others form a multiple-purpose *processor bank*.

3.1.1. Single site

The unit of communication between the processors is the *basic block*, consisting of a source address, a destination address, and a chunk of data. Several end-to-end protocols have been built on top of these blocks. For example, the Single Shot Protocol (SSP) is a simple communication interface to exchange request and reply messages. Services are named by character strings. The name server maintains the location of all services.

When a process wants to make use of some (local) service, it sends a NAME-LOOKUP-REQUEST to the name server to find the location of the service. The name server itself is situated at a well-known address. It sends a reply containing the address of the server back to the original process, using the source address in the basic block that contained the request. Now the process can send requests to the service using SSP-REQUEST messages.

3.1.2. Multiple sites connected by a Wide-Area network

To allow communication with services provided on different rings, rings can be connected by *bridges*. A bridge is a special processor on the ring having its own ring address, or two ring addresses if it connects two rings directly. The bridge serves two functions. First, it helps in locating remote services, and second, it transfers basic blocks between rings. This last property makes the

bridge a data-link level gateway. All this is transparent to the client process. These protocols have been applied successfully in the UNIVERSE Project [Adams82, Leslie84, Wilbur85], which combined several local-area networks using satellite and X.25 wide-area networks.

Remote contact is established as follows. As usual, the process sends a NAME-LOOKUP-REQUEST to the name server. The name of the ring where the server resides is specified in the request. For now we assume that the name server also knows the destination address on that ring. It then sends an ADDRESS-INSERTION-REQUEST to a bridge, which allocates some data structures and acknowledges the request immediately. Then the name server sends a special reply to the client process containing the address of the bridge, and the global address of the server.

The client software then sends an SSP-REQUEST to the bridge, thinking that the bridge is the service. The SSP-REQUEST is automatically converted to a BRIDGE-SSP by inserting the global address of the service. The bridge forwards the BRIDGE-SSP to other bridges using static routing tables, until the destination ring is reached. Each bridge remembers the source address so that a reply can be routed back. The last bridge transforms the BRIDGE-SSP into an ordinary SSP-REQUEST, and sends it to the final destination. Replies are returned using the backward path set up by the bridges, and delivered to the original client process.

The forward path and backward path may now be used by the client process and the server process transparently. When the client thinks it is sending a basic block to the server, it is really sending it to a bridge which forwards it to the destination network. The bridge at the destination network then sends the basic block to the server process which will think that it received the message from the client. The forward and backward paths do not provide any flow control or error correction; this must be taken care of by the end-to-end protocols.

Now suppose that the name server does not know the destination address of the server process. Now the name server sends a REMOTE-NAME-LOOKUP-REQUEST to the name server on the destination ring. The destination address of the name server is fixed on every ring. The local name server sends a request to the remote name server in the same way as a client process would send a request to a remote service. The REMOTE-NAME-LOOKUP-REPLY from the remote name server contains the destination address of the wanted server. The local name server caches the remote address for possible future reuse and sends the client the address of the bridge on the local network in the NAME-LOOKUP-REPLY. Again the client and server are unaware of the bridges between their networks.

3.2. The V-System

The V-System [Cheriton83b] is a distributed operating system running on a collection of processors connected by an Ethernet. The processors are divided into two types: workstations and server machines. The workstations are like processors in the Cambridge Processor Bank, except that workstations have owners that have high-priority access to their machines. The server machines provide services like file access and printing.

3.2.1. Single site

Interprocess communication is through request-reply exchanges: a client process sends a request to a server process, and then awaits the reply subsequently sent by the server. The protocol used here is developed specially for this purpose for optimal performance, and because no suitable standard interprocess communication protocol was available [Cheriton83a].

In V, services are named by character strings. To locate a service, the client *broadcasts* the name of the service over the network (broadcast is a special case of V *multicast*). The service replies with a *process identifier*—a location dependent number that identifies a process—and from thereon contact is established. This scheme has been extended for any kind of object by using directories on each processor, and thus a *global naming directory* was formed.

3.2.2. Multiple sites connected by a Wide-Area network

Internetwork communication in V *should be* implemented as follows. We say "should be," since the current implementation is somewhat simplified. When a process wants to access a remote service, it broadcasts the service name as usual over the local-area network. The gateways on the network forward this message to all the the remote networks, where it is then re-broadcast. The remote service replies as usual, thinking that the gateway on its network is an ordinary client process. This gateway sends the reply back to the local network where the real client resides. The local gateway sends the reply on to the client, which, again, thinks the gateway is the server. The local gateway starts a *local alias process*, a pseudo-process that represents the server. In the same way, the remote gateway starts a *remote alias process* to represent the client. All messages sent between the client and server are forwarded by these alias processes over the wide-area network.

One problem arises since the process identifiers are local to a network. The process identifier of a remote service has no meaning on the local network. To solve this, the process identifiers have to be translated to local process identifiers when passed to the local network. The file server therefore has to be aware of this, by returning the process identifier of the remote alias process, violating transparency.

(The current implementation of the gateway does not support internet broadcasting, making the service locating protocol unusable for locating remote services. Instead, when a process wants to access a remote service, it first has to request the gateway to create a local alias process for the remote service.)

V gateways know about the higher-level protocol, and use this knowledge to optimize communication over wide-area networks (without affecting local networking). For example, if two gateways are connected by a reliable virtual circuit, the gateways can filter out end-to-end acknowledgements, and generate them locally instead. This does not violate end-to-end reliability in V, since all requests are acknowledged by replies in the end. Another optimization done by V gateways is the combination of packets over virtual circuit connections. Furthermore, retransmissions that arrive at a rate too high for the virtual circuit are discarded.

The knowledge of transport protocols makes V gateways at least transport-level gateways. However, the gateways have also higher-level properties. Before forwarding messages, V gateways inspect them to ensure that local security policies are not violated. This way the local network becomes a *security domain*. V gateways thus allow for access control, authentication, and accounting. By implementing these checks in the gateway, instead of at every site, local performance is not affected, nor are the local security policies.

3.3. The Amoeba distributed operating system

The Amoeba Distributed Operating System [Mullender86, Mullender84a, Tanenbaum86] is a research project being carried out at the Vrije Universiteit and the Centrum voor Wiskunde en Informatica, both in Amsterdam. Amoeba is also based on the client-server model. Server processes provide services like file and directory service. Amoeba runs on a collection of Motorola 68010 and 68020 processors connected by 10 Mbit networks.

Processes in Amoeba are addressed by *ports*. Ports are location-independent 48-bit numbers. A process can choose any port it wants to. By taking a random unique 48-bit number, servers can have a private address that they can use on any machine. It is even possible to use the same port for more than one process. This way a service can increase its availability by replicating its server processes. The communication protocol selects one of them.

3.3.1. Single site

A client process invokes a service by sending a request message to the server process. When the server has executed the request it returns a reply message to the client. These request-reply exchanges are called *transactions*[Mullender84b]. They are used as a basis for implementing remote procedure calls.

When a client process starts a transaction, the server has to be located first. This is done by broadcasting a message containing the port of the server process over the local-area network. The machine running the server sends a reply back containing its network address. This information is cached by the client machine, so if it needs the same service in the near future, it can try the same network address without having to broadcast again. If this address turns out to be wrong (servers and their ports can migrate), it can resort to broadcast to locate a server again.

Once the network address of the server is known, a simple protocol

optimized for the local network ensures reliable transmission of the request and reply messages. Moreover, this protocol ensures at-most-once delivery of requests, avoiding problems that occur when requests get executed twice due to retransmissions [Spector82].

3.3.2. Multiple sites connected by a Wide-Area network

Extending the transaction implementation to wide-area networks meets with difficulties. First, the scheme of locating ports by broadcast does not work with multiple Amoeba systems connected by a wide-area network. Wide-area networks usually do not support broadcast. Simulating it by sending the messages separately to each site is expensive, even when minimum spanning trees [Dalal77] are used. Furthermore, a broadcast causes overhead on each site of the wide-area network. Second, the protocol that is efficient for local-area networks is inefficient for wide-area networks. Also, since the only access to the wide-area networks is through high-level interfaces, the low-level Amoeba messages are transported using a high-level protocol.

A solution to the first problem is presented through *publishing*. Servers can publish their port and wide-area network address in the *domains* where they want to be known. A domain is a set of Amoeba sites that are logically related. As soon as a port has *appeared* in an Amoeba site, processes in this site can use the server.

To enable this, a process is created for each port that appears. This process will *stand in* for the remote server by using the port as its own Amoeba address, and is therefore called the *server agent*. If a client process tries to locate the remote server, it will find the server agent instead, and a request message for the server is sent to the server agent. The server agent forwards the request across the wide-area network using the published wide-area network address.

When the request arrives at the remote Amoeba site, a process is created there to send the request message to the server. This process, called the *client agent*, starts a local transaction with the server. The server thinks it received a request from another client process. The reply it returns is sent by the client agent to the server agent. The server agent then forwards the reply to the real client, completing the transaction.

All this is transparent to both the client and server processes. Moreover, it is transparent to the kernels that run the Amoeba transaction protocol. This implies that the transaction protocol is unaware of the existence of the wide-area network, and that no unnecessary overhead exists for local communication. In the same way, the wide-area network protocol knows nothing about transactions, but just forwards complete messages. The client and server agents together form the gateway, which is called a *transformer*[Renesse86]. A transformer is a session-level gateway, since it uses the transport protocol interface. Flow and congestion control are provided by the wide-area network, so the transformer does not have to do it. As it is at a higher level than the network layer it "knows" more about these interactions and can provide valuable information about the distributed processing that crosses network

boundaries.

Publishing is implemented by a server running at each site, together forming the Service for Wide-Area Networks, or SWAN. Each SWAN server listens to a well-known port, and can therefore be easily contacted from anywhere in the world. A port is published by doing a transaction stating port and wide-area network address with each SWAN server in the required domain. Each SWAN server will then automatically start a server agent.

4. Comparison

Having discussed solutions to wide-area networking for several different systems, we will compare their properties in three categories: naming transparency, protocol transparency, and gateway functionality. Each of these categories will be discussed in the following sections.

4.1. Naming transparency

This section discusses whether client processes have to know the location of a service, and whether services have to know the location of the client when passing names in replies. These properties are highly dependent on the local naming strategy.

In the Cambridge system, local services are named by ASCII strings. Names are mapped to local network addresses using a central named server. This scheme is extended to wide-area networking by prefixing the service name with the name of the ring. For example, suppose there are two networks called A and B, each having a printer server called *printer*. A client process on network A can access the local printer using the name *printer*, and the printer server on B using the name *B*printer*. The name server knows which bridge to use to reach B, and the wide-area network address of B. Finding services given their name is not a problem anymore; however, there is no naming transparency. Also, if a process on B wants to pass the name of the printer server to a process on A, it will have to prepend B to the name.

V uses a decentralized naming approach relying on broadcast. In the V-system, the printer server on B should have a different name from the printer server on A, since they do not provide exactly the same service (they use different printers). This scheme is transparent. However, V uses a two-level naming scheme. A process can not send a message to a remote process using the remote process identifier.

Amoeba has an implicit decentralized naming scheme using ports. The port space is not local to a network, and therefore ports can be passed freely from network to network. As in V, the printer servers would have different names. However, since Amoeba gateways do not support broadcast, a port that is passed from A to B has to be published in B first. Since this is different from local operation, Amoeba does not provide full naming transparency either.

4.2. Protocol transparency

In this section we will see to what extent the local protocols are affected by supporting wide-area communication. It is important that the performance of local communication is not degraded, since local communication will represent the bulk of all communication. Changing the local protocol software may not be possible if it is a commercial package without source code. Even if the source code is available, it has to be avoided since it requires all sites on the network to change their system.

The Cambridge system uses the local transport protocols over wide-area networks. This was made possible by using a fixed internetwork packet format (basic blocks). A disadvantage of this approach is that these protocols may not be suitable for wide-area communication. For example, the timeouts that are used in SSP are adjusted for remote services.

V nodes are unaware of gateways, and the local protocol is unaffected. When a remote process is referenced, a local alias process takes care of transfering messages to and from the remote network. As in the Cambridge system, the local protocol was also used for internetwork communication.

Amoeba also provides full protocol transparency, but unlike V, Amoeba does not use the same protocol for wide-area communication. Instead it uses whatever protocol is available on the wide-area network. This means the Amoeba machines can use protocols optimized for the local cast on local networks and the gateways can use other protocols over the wide-area network without the clients and servers knowing about it.

4.3. Gateway functionallity

The different solutions to wide-area communication require different gateway functions. The functionalities of the gateway determine its complexity. The gateway in the systems we discussed have two important tasks. One is to help in locating servers, and the other is to provide transparent communication between processes separated by a wide-area network.

Cambridge bridges help in locating services by forwarding OPEN-REQUESTs. This is a simple operation, using static routing tables. The bridges remember the paths they formed, so that they can use them for forwarding basic blocks. The transport protocols are unknown. Since the bridges just forward the basic blocks, it is possible that for the wide-area network to become flooded or congested, so flow and congestion control may be needed.

V gateways, on the other hand, are fully aware of the transport protocol. For service location, they have to set up minimum spanning trees to forward broadcasts. Gateways optimize the protocol to avoid flooding the wide-area links. Furthermore, gateways check messages to prevent them from violating security constraints. All this makes a V-gateway a complex device.

In Amoeba, the naming and protocol problems are solved separately. The SWAN service takes care that ports are distributed where they are needed. The transformer achieves protocol transparency by transfering complete messages, based on routing information provided by the SWAN. The transformer uses the transport interfaces of both Amoeba and the wide-area networks, and

therefore does not know the transport protocols. Nevertheless the protocols are designed especially for the type of network they are running on. There is not much to be gained in forwarding packets instead of messages if local and wide-area networks differ considerable in bandwidth. Both the SWAN service and the transformer are simple devices.

5. Conclusions

In this paper we discussed how distributed operating systems, designed for local networks, can be connected into a wide-area distributed system. Two problems were identified. One is how to locate a remote service over a wide-area network. The other is providing transparent communication. We described how these problems were solved in the Cambridge Distributed Computing System, the V-System, and the Amoeba Distributed Operating System.

All these systems use special gateways to transfer messages between the local networks. The data-link level gateway used in the Cambridge system supports several transport protocols, but the protocols do not adapt well to wide-area networking. The V transport-level gateway does protocol optimization, since it knows the transport protocol. This makes it better suited for wide-area networking, but it is also a complicated gateway.

The Amoeba gateway uses the transport-level interface of both the local network and the wide-area network, and is therefore a session-level gateway. The gateway is not concerned with how to provide efficient communication, but leaves that problem to the local and wide-area network software. By *standing in* for remote processes, it provides transparent communication without affecting local networking. Services make themselves "known" through *publishing*, that is, they relate their existence to all local networks in which they want to provide their service.

It is argued that the Amoeba gateway provides the same gateway functionality as other gateways, and that the implementation is more efficient. Furthermore, in the normal case when the bandwidth of the wide-area network is considerably lower than that of the local networks, the performance is at least as good as that of other gateways.

Acknowledgements

We would like to thank Gojko Babić, David Holden, Jack Jansen, Bram Janssen, Kari Lang, Martin Turnbull, and Wolfgang Zimmer for valuable dis-

cussions. We also thank Jennifer Steiner for the careful reading of this paper.

REFERENCES

[Adams82]
ADAMS, C. J., ADAMS, G. C., WATERS, A. G., LESLIE, I., and KIRK, P., "Protocol Architecture of the UNIVERSE Project," *Proceedings of the 6th International Conference on Computer Communications*, pp.379-383, September 7-10, 1982.

[Birrell84]
BIRRELL, A. D. and NELSON, B. J., "Implementing Remote Procedure Calls," *ACM Trans. Comp. Syst.*, vol. 2, pp.39-59, Februari 1984.

[Cheriton83a]
CHERITON, D. R., "Local Networking and Internetworking in the V-System," *Proc. Eighth Data Communications Symposium*, October 1983.

[Cheriton83b]
CHERITON, D. R. and ZWAENEPOEL, W., "The Distributed V Kernel and its Performance for Diskless Workstations," *Proc. Ninth ACM Symp. on Operating Systems Principles*, pp.128-140, October 1983.

[Dalal77]
DALAL, Y. K., "Broadcast Protocols in Packet Switched Computer Networks", Ph.D. Dissertation, Computer Science Dept., Stanford University, Stanford, Calif., April 1977.

[Leslie84]
LESLIE, I. M., NEEDHAM, R. M., BURREN, J. W., and ADAMS, G. C., "The Architecture of the Universe Network," *Computer Communication Review*, vol. 14, no. 2, pp.2-9, 1984.

[Mullender84a]
MULLENDER, S. J. and TANENBAUM, A. S., "Protection and Resource Control in Distributed Operating Systems," *Computer Networks*, vol. 8, no. 5,6, pp.421-432, 1984.

[Mullender84b]
MULLENDER, S. J. and RENESSE, R. VAN, "A Secure High-Speed Transaction Protocol," *Proceedings of the Cambridge EUUG Conference*, September 1984.

[Mullender86]
MULLENDER, S. J. and TANENBAUM, A. S., "The Design of a Capability-Based Distributed Operating System," *The Computer Journal*, vol. 29, no. 4, pp.289-300, 1986.

[Needham82]
NEEDHAM, R. M. and HERBERT, A. J., *The Cambridge Distributed Computer System*. Reading, Ma.: Addison-Wesley, 1982.

[Renesse86]
RENESSE, R. VAN and STAVEREN, J. M. VAN, "Wide-Area Communication under Amoeba", IR-117, Dept. of Mathematics and Computer

Science, Vrije Universiteit, Amsterdam, December 1986.

[Spector82]
SPECTOR, A. Z., "Performing Remote Operations Efficiently on a Local Computer Network," *Comm. ACM*, vol. 25, no. 4, pp.246-260, April 1982.

[Sunshine77]
SUNSHINE, C. A., "Interconnection of Computer Networks," *Computer Networks*, vol. 1, no. 3, pp.175-195, January 1977.

[Tanenbaum85]
TANENBAUM, A. S. and RENESSE, R. VAN, "Distributed Operating Systems," *ACM Computing Surveys*, vol. 17, no. 4, pp.419-470, December 1985.

[Tanenbaum86]
TANENBAUM, A. S., MULLENDER, S. J., and RENESSE, R. VAN, "Using Sparse Capabilities in a Distributed Operating System," *Proc. of the 6th Int. Conf. on Distributed Computing Systems*, pp.558-563, May 1986, Vrije Universiteit.

[Wilbur85]
WILBUR, S. R. and KIRSTEIN, P. T., "The Universe Catenet: its Protocols and Issues," *IEEE Proceedings, Part E*, vol. 132, no. 4, pp.189-195, July 1985.

[Zimmermann80]
ZIMMERMANN, H., "OSI Reference Model—The ISO Model of Architecture for Open Systems Intercon nection," *IEEE Trans. Comm.*, vol. COM-28, pp.425-432, April 1980.

Applications

A Distributed, Parallel, Fault Tolerant Computing System

Henri E. Bal
Robbert van Renesse
Andrew S. Tanenbaum
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Distributed systems offer two principal advantages over centralized ones: higher computing speed through the use of many computers running in parallel, and higher reliability through redundancy. This paper describes how the Amoeba distributed system meets these goals. In particular it describes Amoeba and how two important classes of algorithms, branch and bound and alpha-beta search, can be run in a parallel, fault-tolerant way on Amoeba. The results of some experiments comparing these algorithms on a single processor and on Amoeba are also discussed.

1. Introduction

Distributed computing systems have two principal advantages over traditional centralized ones: speed and reliability. First consider speed. As computing technology advances, it becomes increasingly difficult and expensive to make computers faster by just increasing the speed of the chips. Electrical signals in copper wire only travel at 2/3 the speed of light, or about 20 cm/nanosecond, so very fast computers must be very small, which leads to severe heat dissipation problems among other things. The obvious solution is to harness together a large number of moderately fast computers to achieve the same computing power as one very fast computer, but at a fraction of the cost.

The second big advantage of distributed computing systems is the reliability that can be achieved by using a large number of processors. If a system consists of 100 processors and 1 of them malfunctions, the system should be able to continue running with just a one percent loss in performance. Furthermore, if the system is well-designed, when a processor crashes, this event should be detected and recovered from without ruining the computation that was in

A Distributed, Parallel, Fault Tolerant Computing System
H.E. Bal, R. van Renesse, and A.S. Tanenbaum
Informatica Report IR-106
Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam
October 1985

progress at the time of the failure. How this is achieved in the Amoeba system will be described later in this paper.

Many ways of organizing multiple processors into distributed systems have been proposed. At one end of the spectrum are the *loosely-coupled systems* consisting of a number of independent computers, each with its own operating system and users, exchanging files and mail over a public data network. At the other end of the spectrum are *tightly-coupled systems* with multiple processors on the same bus and sharing a common memory. In between are systems consisting of mini- or microcomputers communicating over a fast local network and all running a single, system-wide operating system. This paper describes a system in the latter category that can take advantage of a large number of microprocessors working together on a single problem, and also has a high degree of fault tolerance.

2. The Amoeba system

This system, called Amoeba [Mullender86, Mullender84, Mullender85], consists of a collection of (possibly different) processors, each with its own local memory, which communicate over a local network. Currently, we use mainly Motorola 68010 processors connected by a 10 Mbps token ring (Pronet), although Amoeba also runs on the VAX, NS16032, PDP-11 and IBM-PC. Amoeba is composed of four basic components. First, each user has a personal workstation, to be used for editing on a bit-map graphics terminal and other activities that require dedicated computing power for interactive work. Second, there is a pool of processors that can be dynamically allocated to users as needed. For example, a user who wants to run a 5-pass compiler might be allocated 5 pool processors for the duration of the compilation, to allow the passes to run largely in parallel. Third, there are specialized servers: file servers, directory servers, process servers, bank servers (for accounting) etc. Fourth, there are gateways that connect the system to similar systems elsewhere.

The amoeba software is based on objects protected by capabilities. Each file, directory, process, bank account, etc. can be viewed as an object (an abstract data type) on which operations (e.g., READ, DELETE) can be performed by the process that manages that object. When an object is created, a capability for it is given to the object's owner. To perform any operation on an object, the capability must be presented. Capabilities are protected cryptographically, and are managed directly by user programs.

A process or *cluster* in Amoeba consists of one or more *tasks* that share a common address space and run in parallel. Several independent clusters may run on a single workstation or pool processor. Tasks communicate using a simple form of remote procedure call: the client sends a request to any server who is willing to offer a certain service and some server sends a response back to the client. While a task is waiting for a response, it is blocked and cannot continue computing, although other tasks in its cluster may run if they have work to do. This scheme is much simpler and vastly more efficient than the ISO OSI 7-layer Reference Model [Zimmermann80].

3. Parallel algorithms on Amoeba

In this section we will provide a brief overview of how heuristic search algorithms have been programmed in parallel in a fault-tolerant way on Amoeba. Heuristic search is a technique for finding a feasible or (sub) optimal solution to a given problem when the set of candidate solutions is very large. One typical problem is the Traveling Salesman problem, in which it is desired to find the cheapest route for a salesman to visit each of the n cities in his territory exactly once. Since there are $(n-1)!$ possible routes, for large n, it is not possible to examine all of them and then take the best one. Playing chess is another example of a problem with a large search space.

One way to approach this kind of problem on Amoeba is is to allocate $k+1$ pool processors to work on the problem. As a simple example, to solve the 10-city traveling salesman problem starting from London, one could allocate nine processors and have processor 1 examine all paths starting London-Amsterdam, processor 2 examine all paths starting London-Zurich, etc. Processor 1 would then allocate eight more processors, giving the first one the partial path London-Amsterdam-Zurich, the second one the partial path London-Amsterdam-Paris, etc.

Since there will never be enough processors available, at some point a processor will itself have to evaluate the best full path starting with the partial path given to it, rather than "subcontracting" the work out. When a processor has discovered the best total path achievable with the partial path it was given, it reports that back to the processor that invoked it. When the invoking processor has collected all the results from its "subcontractors," it chooses the best one and reports that back. When the results have gotten back to the top level, the initial processor selects the best one, and the problem has been solved. If a subcontractor crashes (i.e. fails to respond within a specified time to enquiries of the form "are you still working on the problem?") then the processor requesting the work finds a new subcontractor to do the work. In the following sections, we will describe more sophisticated strategies (one for traveling salesman type problems and one for game playing), along with their implementations and some empirical results.

4. Parallel branch and bound on Amoeba

The branch-and-bound method is a technique for solving a large class of combinatorial optimization problems. It has been applied to Integer Programming, Machine Scheduling problems, the Traveling Salesman Problem (TSP), and many others [Lawler66]. Abstractly, the method uses a *tree* to structure the space of possible solutions. A *branching rule* tells how the tree is built. For the TSP, a node of the tree represents a partial tour. Each node has a branch for every city that is not on this partial tour. Figure 1 shows a tree for a 4-city problem. Note that a leaf represents a full tour (a solution). For example, the leftmost branch represents the tour London - Amsterdam - Paris - Washington.

A *bounding rule* avoids searching the whole tree. For TSP, the bounding rule is simple. If the length of a partial tour exceeds the length of any already

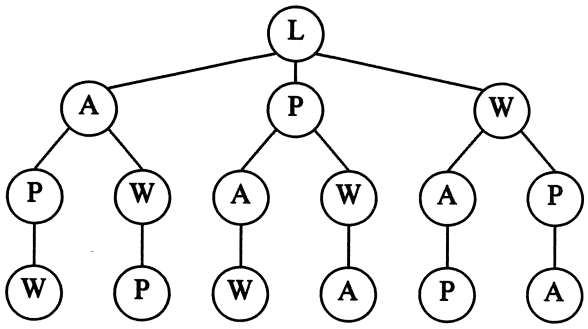


FIGURE 1. Tree for a 4-city Traveling Salesman Problem for London, Amsterdam, Paris, and Washington.

known solution, the partial tour will never lead to a solution better than what is already known. For example, if the 6-city tour London - Paris - Amsterdam - New York - Boston - Washington has already been found to be 8630 km, then partial tours starting London - New York - Paris (length 11850 km) cannot possibly be better than the best tour already found. Efficient branch-and-bound algorithms aim at finding a nearly-optimal solution at an early stage, making pruning as effective as possible. A good heuristic for TSP is to try the nearest city first.

Parallelism in a branch-and-bound algorithm is obtained by searching parts of the tree in parallel. If enough processors were available, a new processor could be allocated to every node of the tree. Every processor would select the best partial path from its children and report the result back to its parent. If there are N cities, this approach would require O(N!) processors. More realisticly, the work has to be divided among the available processors. In our model, each processor traverses a part of the tree, up to a certain *depth*, hands out the subtree below that node to a 'subcontractor', and continues with the rest of its own subtree. figure 2 shows how the tree of figure 1 can be searched, using a 2-level processor hierarchy (i.e., a subcontractor has no subcontractors itself).

The processor that traverses the top part of the tree (the root processor) searches one level. It splits off three subtrees of depth two each, that are traversed by subcontractors. This algorithm is shown in figure 3. The algorithm sets the global variable 'minimum' to the length of the shortest path. This variable is pre-initialized with a very high value.

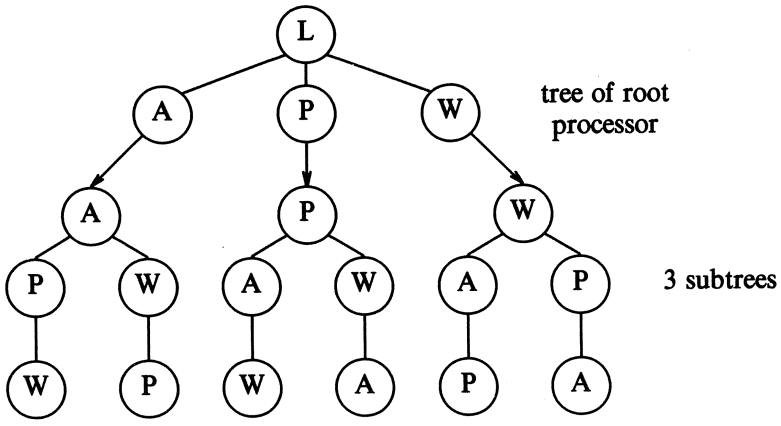


FIGURE 2. Example of a distributed tree search

```
procedure traverse(node,depth,length);
begin
    { 'length' is the length of the partial path so far.
      'depth' is the number of levels to be searched before
      the rest of the tree should be handed out to a subcontractor }
    if length < minimum then  { pruning if length >= minimum }
    begin
        if 'node' is a leaf then
            minimum := length;
        else if depth = 0 then
            hand out subtree rooted at 'node' to a subcontractor;
        else
            for all children c of 'node' do
                traverse(c,depth - 1,length+distance(node,c));
    end
end
```

FIGURE 3. Tree traversal algorithm

A processor only blocks if it tries to hand out a subtree while there are no free subcontractors. Each subcontractor executes the same traversal process, with a different initial node and probably with a different initial depth.

The Traveling Salesman Problem has been implemented under Amoeba using the algorithm described above. The client/server model advocated by Amoeba was found to be very suitable for this algorithm. For simplicity, the implementation uses only a 2-level processor hierarchy.

A subcontractor can be viewed as an Amoeba *server* process (cluster). The service it offers is the evaluation of a TSP subtree. Each server repeatedly waits for some work, performs the work, and returns the result. The root processor is a *client* process (see figure). The 'handing out of work' is

implemented using Amoeba *transactions*. Concurrency within the client process is achieved by having a separate task (as defined in section 2) in the client cluster for every server. This *job server* task controls the communication with one specific server. If the client wants to hand out some work, it tries to do a transaction with a job server. If there is a free job server, this job server will accept the transaction, return an acknowledgement to the client, and then do a transaction with its server. The job server passes a partial path and the current best solution to the server. When the server finishes the evaluation of the subtree, the transaction finishes and the job server is unblocked. The job server checks if it has to update the current best solution and then becomes available for the next request. The client proceeds as soon as it receives the acknowledgement. The entire client cluster only blocks if all job server tasks are blocked (i.e., if all servers are busy) and the client tries to do a transaction with a job server.

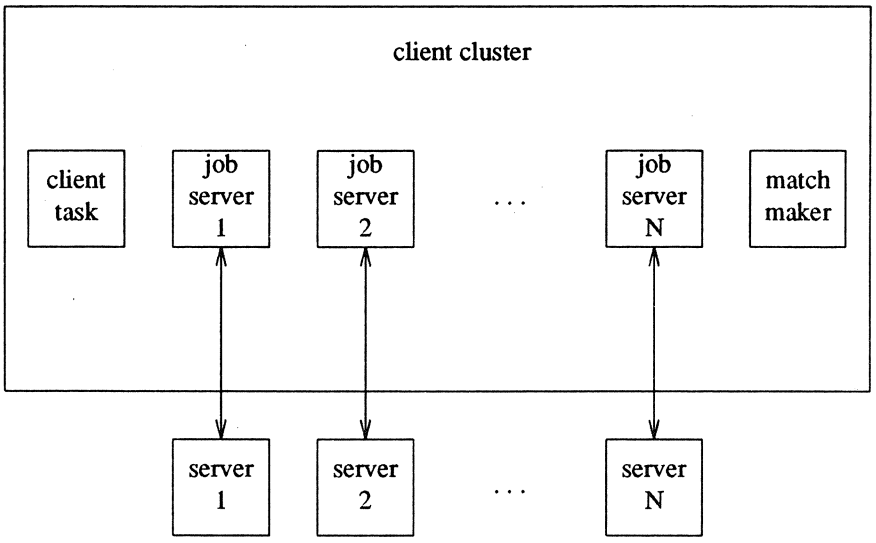


FIGURE 4. Process structure of the TSP program

Of special importance is the way servers join and leave the system. Whenever a new server is started, this server reports itself to a special *matchmaker* task that is also part of the client cluster. This matchmaker task creates a job server task for the server and from then on the server can participate in the game. So extra processors can be added at any time to speed up the program.

The job server mechanism is also used to achieve a high degree of fault tolerance. During transactions, the Amoeba kernel of the client sends "are-you-still-there?" messages to the kernel of the server at regular intervals. If the kernel of the server does not respond within a certain time interval, the transaction is aborted. The job server notes that the transaction has failed and concludes that its server processor has crashed. It hands out its work to any other job server. Once this work has been accepted, the job server stops

executing. The crashed server processor no longer participates in the game. When it is brought back up, it reports itself to the matchmaker as described above, to register its availability for doing work, at which time a new job server task is created to handle it. Since the client task, job server tasks, and matchmaker task are all part of the same address space, the inter-task communication is highly efficient.

Although fault-tolerance may not be of vital importance to a TSP program, it is a useful feature to have, especially as it is almost for free. (The entire implementation of fault-tolerance in the TSP program takes only a few lines of code). For example, if some Amoeba user is going out for lunch, the processor of his workstation can be used by someone else to speed up his program. When the owner of the workstation comes back, he can blindly kill the foreign process without disrupting the overall program.

This model still has one Achilles heel. A failure in the client processor cannot be recovered from easily, as no one may detect the fault. For the application above it will be sufficient to run the client on a processor that no one will take away. For more critical applications, the "boot service" can be used to keep an eye on the root processor, just as the root processor keeps an eye on the server processors. Any process can register with the boot service, which then polls it periodically. If the registered process fails to respond to polls, the boot service reboots the process on a different processor.

5. Parallel Alpha-Beta search on Amoeba

Alpha-beta search is an efficient method of searching game trees for two-person, zero-sum games. A node in such a game tree corresponds to a position in the game. Each node has one branch for every possible move in that position. A value associated with the node indicates how good that position is for the player who is about to move (let's assume this player is 'white'). At even levels of the tree, this value is the *maximum* of the values of its children; at odd levels it is the *minimum*, as the search algorithm assumes black will choose the move that is least profitable for white. Most implementations inverse the values of the odd level nodes, so the values are maximized at all levels.

The alpha-beta algorithm finds the best move in the current position, searching only part of tree. It uses a *search window* (alpha,beta) and prunes positions whose values fall outside this window. The algorithm is shown in figure 5.

Alpha-beta search differs significantly from branch-and-bound in the way the best solution is constructed. A branch-and-bound program (potentially) updates its solution every time a processor visits a leaf node (see figure 3). That processor only needs to know the current best solution and the value associated with the leaf. An alpha-beta program, on the other hand, has to *combine* the values of the leaves and the interior nodes, using the structure of the tree. Some parallel alpha-beta programs realize this by having a dedicated processor for every node (up to a certain level) that collects the results of the child processors [Finkel82]. As a disadvantage of this approach, processors

```
function AlphaBeta(node,depth,alpha,beta): integer;
begin
    if depth = 0 then
        AlphaBeta := evaluation(node)
    else
      for all children c of 'node' do
      begin
          r := - AlphaBeta(c,depth - 1,- beta,- alpha)
          if r > alpha then
          begin
              alpha := r;
              if alpha >= beta then
                  exit loop;   -- pruning
          end
      end
    AlphaBeta := alpha
end
```

FIGURE 5. Sequential alpha-beta algorithm

associated with high level interior nodes spend most of their time waiting for their children to finish.

Our solution avoids this problem by working the other way round. The child processors compute the values for their parent nodes, so there is no need for their parent processors to wait. This is illustrated in figure 6. In figure 6(a), the subtrees rooted at nodes 3,4,6, and 7 have been evaluated. After some subcontractor has evaluated the subtree rooted at node 8, the value of the parent of node 8 (node 5) is updated (as 20 > 15). This is shown in figure 6(b). Furthermore, the evaluation of the subtree rooted at 5 has now been completed. As its final value (-20) is the highest value of level 1 (-20 > -30), the value of node 1 is updated too.

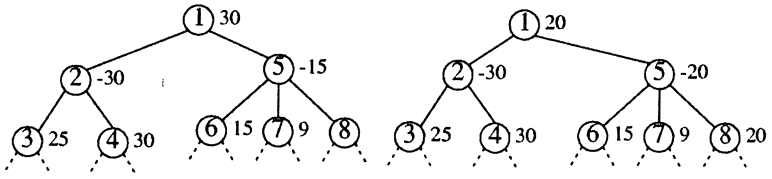


FIGURE 6. Example of alpha-beta search

Clearly, the child processors need information from other processors to compute these values. We store all information in an *explicit* tree structure, so the search tree is no longer just a concept, but it is actually built as a data structure. This tree is distributed over all processors, each processor containing the part of the tree it works on.

With this approach we can use basically the same tree traversal algorithm and the same process structure as for TSP. The only difference is that TSP

updates a single global solution after evaluating a leaf and alpha-beta updates the values of the ancestor nodes of the leaf.

Each node also contains the alpha and beta bounds for its subtree. After the value of a node has been improved (as a result of evaluating a leaf) this new value can be used as a tighter alpha bound for its children. Each child can use this new alpha value as a tighter beta bound for its own children, and so on. So new values are propagated down the tree, to ensure each node uses the smallest possible alpha-beta window. In principle, new bounds can even be propagated across processor boundaries. However, this would also increase the communication overhead. We have not yet implemented this kind of propagation.

6. Discussion

We have done some measurements on the TSP and the alpha-beta programs. The hardware used was a collection of 10 MHz 68010 CPU's connected by a 10 Mpbs token ring. For each program, we ran both a sequential (single processor) version and a parallel (multi-processor) version. Each parallel version uses one processor for the client process and a varying number of processors for the servers. Note that with only one server, there is still some parallelism, as the client can find the next subtree to hand out, while the server is working on the previous subtree.

The depths of the subtrees are important parameters of the TSP algorithm. If the client processor distributes work at a too high level, the effectiveness of pruning will be severely weakened. For example, if it traverses just one level, then the best solution in the leftmost branch of the tree cannot be used as a bound in its neighbor branch, as these branches are searched simultaneously. Increasing the depth of the root subtree will decrease this effect, at the cost of more communication between the root processor and its subcontractors. To achieve high performance, a good compromise has to be found. For an 11-city problem we found the optimal search depth of the client to be three levels. The results for an 11-city problem using this search depth are shown in table 6.1. The last entry in the table shows the speedup over the 1-server version. With 7 processors (1 client and 6 servers) a 5-fold speedup over the sequential program is achieved.

version	time(secs)	speedup
sequential	637.2	
1 server	548.1	1
2 servers	309.7	1.77
3 servers	218.2	2.51
4 servers	171.7	3.19
5 servers	141.5	3.87
6 servers	124.2	4.41

Table 6.1. Results for 11-city Traveling Salesman Problem.

To measure the performance of the alpha-beta algorithm, we implemented the game of *Othello*, using this algorithm. Table 6.2 shows the time to evaluate a position, averaged over five different positions with a fan-out (number of moves) of approximately fifteen. The depth of the search tree was four plies. As for TSP, the division of labour between the client and the servers is important. For the parallel versions the client searched three plies, the servers searched one ply.

version	time(secs)	speedup	#evaluations	search overhead
sequential	266.9		2670	1
1 server	324.6	1	2670	1
2 servers	196.2	1.65	3925	1.47
3 servers	153.3	2.12	4732	1.77
4 servers	125.1	2.59	5676	2.13
5 servers	114.0	2.84	6424	2.40
6 servers	111.5	2.91	6719	2.51

TABLE 6.2. Results for Othello implementation of alpha-beta search.

The results show that the speedup achieved is significantly better for TSP than for alpha-beta search. The main reason is that alpha-beta search suffers more from the decrease in pruning efficiency than TSP. The third entry in table 6.2 shows the number of leaves visited by alpha-beta (i.e., the number of static evaluations). This number is a yardstick for the total amount of work done. The last entry shows the search overhead over the sequential version.

Several other authors have studied parallel branch-and-bound algorithms [Finkel85, Wah85, El-Dessouki80, Lai83, Lai84] and parallel alpha-beta search algorithms [Wah85, Marsland82, Finkel82, Finkel83, Akl80, El-Dessouki84]. Good surveys on multiprocessing of combinatorial search problems in general and of parallel game tree search can be found in [Wah85] and [Marsland82] respectively.

Finkel and Manber [Finkel85] use a distributed computing system, CRYSTAL, similar to the Amoeba system. CRYSTAL consists of VAX 11/750 computers connected by a token ring. They implemented a distributed backtracking/branch-and-bound package (DIB) with a clean, sequential, user interface that relieves the programmer of the burdens associated with parallel programming. As a disadvantage, the user has little control over the order of the tree traversal, which was shown to be important.

Early parallel alpha-beta algorithms [Finkel82] aimed at minimizing communication costs, but more or less overlooked the problem of decreased pruning efficiency. Akl et. al [Akl80]. proposed the idea of searching the tree in two phases. During phase 1 only those nodes that cannot possibly be pruned (the minimal tree) are searched. In phase 2, where the rest of the tree is searched, pruning will be highly effective. Finkel and Fishburn [Finkel83] reported a revised implementation of their original algorithm using this "mandatory work first" technique. Their analysis shows a significant improvement

for strongly ordered trees. A practical inconvenience is the fact that the tree has to be searched twice, so part of it probably has to be generated twice.

An alternative proposed by Campbell [Marsland82] is the Principal Variation Search. This algorithm aims at minimizing the number of nodes to be searched, at the cost of some processor idle time. Also, it assumes a hierarchical processor architecture.

Moser [Moser84] has implemented tree splitting in his chess program WATchess 3.0. Although he only uses tree splitting at the highest level of the tree (i.e. after one move), he achieves a good speedup, due to the use of aspiration search.

Our implementations of TSP and alpha-beta search have been deliberately kept simple initially, as we implemented them just to gain some experience with programming under Amoeba. However, our results so far have given us all faith that the primitives offered by Amoeba are sufficiently general for more advanced implementations.

In the near future we will study the implementation of other applications. Among the applications that may be suitable for a distributed system are divide-and-conquer algorithms [Horowitz83], simulation [Bryant79, Bezivin82, Christopher82, Jefferson85], matrix problems [Wise85], design automation [Rutenbar84], compilation [Miller82], and AI problem solving [Smith80].

References

[Akl80]
AKL, S.G., BARNARD, D.T., and DORAN, R.J., "Design, Analysis, and Implementation of a Parallel Alpha-Beta Algorithm", Report 80-98, Queen's University, Kingston, Canada, April 1980.

[Bezivin82]
BEZIVIN, J. and IMBERT, H., "Adapting a Simulation Language to a Distributed Environment," *Proc. 3th Int. Conf. on Distributed Computing Systems*, pp.596-603, October 1982.

[Bryant79]
BRYANT, R.E., "Simulation on a Distributed System," *Proc. 1st Int. Conf. on Distributed Computing Systems*, pp.544-552, October 1979.

[Christopher82]
CHRISTOPHER, T., EVENS, M., GARGEYA, R.R., and LEONHARDT, T., "Structure of a Distributed Simulation System," *Proc. 3th Int. Conf. on Distributed Computing Systems*, pp.584-589, October 1982.

[El-Dessouki80]
EL-DESSOUKI, O.I. and HUEN, W.H., "Distributed Enumeration on Between Computers," *IEEE Trans. on Computers*, vol. C-29, no. 9, pp.818-825, September 1980.

[El-Dessouki84]
EL-DESSOUKI, O. I. and DARWISH, N., "Distributed Search on Game Trees," *Proc. 4th Int. Conf. on Distributed Computing Systems*, pp.183-

191, May 1984.

[Finkel85]
FINKEL, R. and MANBER, U., "DIB - A Distributed Implementation of Backtracking," *Proc. 5th Int. Conf. on Distributed Computing Systems*, pp.446-452, May 1985.

[Finkel82]
FINKEL, R.A. and FISHBURN, J.P., "Parallelism in Alpha-Beta Search," *Artificial Intelligence*, vol. 19, pp.89-106, 1982.

[Finkel83]
FINKEL, R.A. and FISHBURN, J.P., "Improved Speedup Bounds for Parallel Alpha-Beta search," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. PAMI-5, no. 1, pp.89-92, January 1983.

[Horowitz83]
HOROWITZ, E. and ZORAT, A., "Divide-and-Conquer for Parallel Processing," *IEEE Trans. on Computers*, vol. C-32, no. 6, pp.582-585, June 1983.

[Jefferson85]
JEFFERSON, D.R., "Virtual Time," *TOPLAS*, vol. 7, no. 3, pp.404-425, July 1985.

[Lai84]
LAI, N. and MILLER, B.P., "The Traveling Salesman Problem: The Development of a Distributed Computation", Report 84.17, University of California at Berkeley, December 1984.

[Lai83]
LAI, T.H. and SAHNI, S., "Anomalies in Parallel Branch-and-Bound Algorithms," *Proc. of the 1983 Int. Conf. on Parallel Processing*, pp.183-190, August 1983.

[Lawler66]
LAWLER, E.L. and WOOD, D.E., "Branch-and-bound Methods: a survey," *Operations Research*, vol. 14, no. 4, pp.699-719, July 1966.

[Marsland82]
MARSLAND, T.A. and CAMPBELL, M., "Parallel Search of Strongly Ordered Game Trees," *Computing Surveys*, vol. 14, no. 4, pp.533-551, December 1982.

[Miller82]
MILLER, J.A. and LEBLANC, R.J., "Distributed Compilation: A Case Study," *Proc. 3th Int. Conf. on Distributed Computing Systems*, pp.548-553, October 1982.

[Moser84]
MOSER, L., *An Experiment in Distributed Game Tree Searching*. University of Waterloo, 1984.

[Mullender84]
MULLENDER, S. J. and TANENBAUM, A. S., "Protection and Resource Control in Distributed Operating Systems," *Computer Networks*, vol. 8, no. 5,6, pp.421-432, 1984.

[Mullender85]

MULLENDER, S. J. and TANENBAUM, A. S., "A Distributed File Service Based on Optimistic Concurrency Control," *Proceedings of the 10th Symposium on Operating Systems Principles*, pp.51-62, December 1985.

[Mullender86]
MULLENDER, S. J. and TANENBAUM, A. S., "The Design of a Capability-Based Distributed Operating System," *The Computer Journal*, vol. 29, no. 4, pp.289-300, 1986.

[Rutenbar84]
RUTENBAR, R.A., MUDGE, T.N., and ATKINS, D.E., "A Class of Cellular Architectures to Support Physical Design Automation," *IEEE Trans. on Computer-Aided Design*, vol. CAD-3, no. 4, pp.264-278, October 1984.

[Smith80]
SMITH, R. G., "The Contract Net Protocol: High-level Communication and Control in a Distributed Problem Solver," *IEEE Trans. on Computers*, vol. C-29, no. 12, pp.1104-1113, December 1980.

[Wah85]
WAH, B.W., LI, G.-J., and YU, C.F., "Multiprocessing of Combinatorial Search Problems," *IEEE Computer*, vol. 18, no. 6, pp.93-108, June 1985.

[Wise85]
WISE, D.S., "Representing Matrices as Quadtrees for Parallel Processors," *Inf. Proc. Letters*, vol. 20, pp.195-199, May 1985.

[Zimmermann80]
ZIMMERMANN, H., "OSI Reference Model—The ISO Model of Architecture for Open Systems Intercon nection," *IEEE Trans. Comm.*, vol. COM-28, pp.425-432, April 1980.

Parallel and Distributed Compilations in Loosely-Coupled Systems: A Case Study

Erik H. Baalbergen
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

One application of large-grain parallelism is the use of parallel and distributed compilations by *make*, running under UNIX.† The original version of *make* executes its compilation commands successively. 'Making' a large system could therefore take a large amount of time. An increase in efficiency may be achieved by a parallel version of *make*, which tries to execute the compilations simultaneously. A parallel, non-distributed, version of *make* turns out to be inefficient. The compilations, which are mainly cpu-bound, slow each other down due to degradation of the processor's performance. A solution may be found in the idea of boarding out (part of the) compilations to other processors. This resulted in a study of how to do compilations in a distributed manner.

The aspect of having a system of loosely-coupled processors is an important issue in the field of distributed compilations. The relatively high cost of doing transactions (compared to local actions) in a loosely-coupled system makes the use of low-level inter-processor communication (e.g., the execution of system calls on another processor) inefficient. A UNIX network system like the *Amoeba Connection* turns out to be unsuitable for doing distributed compilations. It is shown that much overhead results from the communication between the system that contains the source code to be compiled and the system that does the compilation. Another possibility is to copy the source code to the other processor's data space, execute a local compilation on that processor and send the results back; this greatly reduces the communication overhead. The time needed to send the source to and receive the code from the remote processor is negligible compared with the overhead mentioned earlier.

In order to create a parallel and distributed *make*, I adapted the original 'make' program by adding a module for finding out which compilations can be executed in parallel, depending on the actions to be taken, the actions already finished, and the present files. Furthermore, I created various versions of the UNIX C compiler *cc* in order to perform some measurements.

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

Parallel and Distributed Compilations in Loosely-Coupled Systems
ERIK H. BAALBERGEN
Proc. of workshop "Large Grain Parallelism"
Providence, RI
Oct. 1986

1. INTRODUCTION

The availability of networks of personal workstations has increased interest in parallel and distributed compilations. A decrease of the response time is the main motive for executing compilations in parallel on several processors. This paper is a description of an experiment set up to examine the feasibility of using more than one processor for doing compilations. The experiment took place in a UNIX environment and consisted of two parts: creating two different distributed versions of the C compiler *cc* and constructing a version of the UNIX tool *make* [Feldman78] that could run its compilations in parallel. The aspect of being tightly-coupled or loosely-coupled turns out to be an important issue in determining whether a specific network is suitable for doing distributed compilations.

The test configuration consisted of four VAX 11/750s, each of them running UNIX version 4.1 BSD. The machines are connected by a 10 Mbps token ring (Pronet). Pronet was made available to user programs by incorporating the **Amoeba** 3.0 [Tanenbaum81] communication primitives into UNIX [Renesse84].

A distributed compilation can be done in various ways. One possibility is to create a version of the compiler in which the system calls may be carried out remotely. This can be achieved by using a UNIX network system like the *Amoeba Connection,* similar to the *Newcastle Connection* [Brownbridge82]. The system combines the file systems of each of the connected machines by allowing access to files and execution of programs on other systems. The compiler runs remotely (i.e., on another processor) but each system call concerning the source code should be executed on the processor that runs the file system of the source code. Another possibility is to isolate components from the compiler and execute some of them remotely. One problem with remote compilations is that the compiling program should produce code for the local machine. Each of the connected machines should therefore have a compiler for each of the other machines. This is no problem if the connected machines are similar, as is the case in the test environment. Another problem arises in the second kind of compilation: libraries and included source code should be derived from the source machine. This, too, caused no problem in our test environment, as will be shown.

It must be said that many of the results depend strongly on our configuration, especially with regard to the communication overhead. Results from a common network operation, performed in the same configuration, are included to give an indication of the overhead.

2. AMOEBA AND THE AMOEBA CONNECTION

Amoeba is a distributed operating system developed at the Vrije Universiteit [Tanenbaum81]. The **Amoeba** communication primitives are described in [Tanenbaum84]. **Amoeba** uses a "request-reply" or "transaction" style of communication, in which the basic primitive is the client sending a request to a server and the server sending a reply back to the client. Such a pair of request and reply messages is henceforth called a *transaction*. The implementation of the primitives in UNIX created the possibility for two processes running on

different systems to communicate by means of transactions. This has been exploited in various user programs such as copying files between various systems, remote execution of commands, sharing of resources and remote logging in [Renesse84]. An application of the transaction mechanism is a UNIX system-call server. A process on machine *A* can ask a system-call server on machine *B* to execute a system call, such as **open**, **read**, or **write**. The strategy used to implement the remote system calls is to build an extra layer on the kernel. A program does not directly invoke the kernel but calls a stub routine which checks whether the command must be done locally or remotely. Local commands are passed directly to the kernel. Remote commands are passed to a system call server on the proper machine by doing a transaction with the system-call server. A great advantage of the use of this extra layer is that existing programs need not be rewritten or even recompiled. They only have to be relinked with a library of stub routines. The naming scheme for remote files (i.e., files on other UNIX systems), the system-call server and the stub-routine library together form the *Amoeba Connection.* The connection was found useful in our experiment, although the overhead was large.

The following table gives an indication of the speed of the connection in terms of response time, measured in seconds. Three versions of the UNIX file-copy command *cp* are compared: plain *cp*, able to copy files on the local machine only; *rcp*, the inter-machine file-copy program as described in [Renesse84]; and *fcp*, which is plain *cp* linked with the *Amoeba Connection* library. 'Local' is a file copy from one disk to another on the same system. 'Remote' is a file copy from the local system to another system. All measurements took place on lightly loaded machines.

number of bytes	cp	rcp		fcp	
	local	local	remote	local	remote
1	0.28	1.08	3.55	.25	0.40
1024	0.28	1.20	3.60	0.25	0.43
10240	0.40	1.93	3.95	0.38	0.83

TABLE 1.

3 THE EXPERIMENT AND ITS RESULTS

3.1. A distributed compiler

The first phase of the experiment was to construct two distributed versions of *cc* which is the UNIX C compiler. *Cc* is a program that causes C source code to be passed through several compilation programs. The first step is done by the C preprocessor *cpp* which performs macro substitution, file inclusion and elimination of source code, depending on several user-specified, preprocess-time conditions. Next follows the compiler proper, *ccom*, which is a two-pass portable C compiler [Johnson79]. The assembly code generated by the two-pass compiler is translated to object code by the UNIX assembler *as*. The program *ld*, which is the UNIX link editor, finally combines the object programs,

together with some libraries, into one program which may be executed. The first distributed version of the C compiler is based on the *Amoeba Connection.* The four compilation programs are relinked to allow files on other systems to be compiled. The resulting compiler, together with its driver *cc*, is installed on each of the connected machines. (Having multiple copies of the compiler is in fact an optimization with regards to our implementation of the *Amoeba Connection*; remote execution is allowed only if the program is situated in the file system belonging to the remote processor.) The remote compilation of a source file on **A** is now done jijwfuoking *cc* on some otjijwfuoking *cc* on some otst phase of the experiment was to construct two distributed versions of *cc* which is the UNIX C compiler. *Cc* is a program that causes C source code to be passed through several compilation programs. The first step is done by the C preprocessor *cpp* which performs macro substitution, file inclusion and elimination of source code, depending on several user-specified, preprocess-time conditions. Next follows the compiler proper, *ccom*, which is a two-pass portable C compiler [Johnson79]. The assembly code generated by the two-pass compiler is translated to object code by the UNIX assembler *as*. The program *ld*, which is the UNIX link editor, finally combines the object programs, together with some libraries, into one program which may be executed. The first distributed version of the C compiler is based on the *Amoeba Connection.* The four compilation programs are relinked to allow files on other systems to be compiled. The resulting compiler, together with its driver *cc*, is installed on each of the connected machines. (Having multiple copies of the compiler is in fact an optimization with regards to our implementation of the *Amoeba Connection*; remote execution is allowed only if the program is situated in the file system belonging to the remote processor.) The remote compilation of a source file on **A** is now done ults for response times (compared to plain *cc*) are listed below.

number of *C* source lines	local using cc	Amoeba Connection		Compiler server	
-	-	local	remote	local	remote
20	3.27	4.05	21.45	3.07	5.58
200	9.17	9.82	30.57	9.07	11.33
2000	68.75	68.68	105.28	68.98	69.63

TABLE 2.

Table 2 shows that remote compilation using the compiler server gives a better response time than using the compiler, based on the Amoeba Connection.

3.2. Distributed and parallel 'make'

The second part of the experiment was to apply the result of the first part to the UNIX program *make*. Many programs developed under UNIX consist of a set of C source files which have to be compiled. *Make* performs the compilations sequentially. Doing the compilations in parallel on the same processor is not a solution, as table 3 shows. A solution is found in parallel and distributed execution of the compilations. Table 3 shows results of the three ways of doing independent compilations (i.e., no compilation uses results of the other compilations.) The file to be compiled was about 1800 lines. The maximum number of available processors was 4. The compiler server is used for doing the distributed compilations.

number of compilations	locally sequentially	locally in parallel	distributed in parallel
1	68.67	-	-
2	137.34	127.32	69.58
3	206.00	183.45	73.22
4	274.66	242.50	78.35

TABLE 3.

Adapting *make* in order to execute compilations in parallel on several processors was done easily. *Make* maintains a list of programs that can also run on other machines. A command is executed in parallel to the other commands under the following conditions: the program appears on the distributed-program list; the necessary files are present; the compilations on which the command depends are already done; and there is still enough remote processing power. The only program currently in the distributed-program list is *cc*, but other compilers and translating programs may be added to this list. The processing power of a machine is computed by keeping an account of the number of compilations started by *make* on that machine.

4. PLANS

From the experiment we learned that splitting up compilations and running the environment and machine independent phases on other processors results in a remarkable increase in response time, especially if several sources need to be compiled. Splitting up a single compilation is also a result of the philosophy behind the **Amsterdam Compiler Kit (ACK)**, a project at the Vrije Universiteit in the area of compiler construction [Tanenbaum83]. A compilation in **ACK** causes a program to pass through several components: a front end, which translates the program into machine-independent intermediate code; a peephole and global optimizer; a back end, which translates from intermediate code into assembly code; an assembler; and a linker. The idea is to have a pool of processors, each of them running a dedicated server performing one of the components. **ACK** tries to allocate a server for each of the phases and passes the program through the pipeline of servers.

REFERENCES

[Brownbridge82]
D.R. BROWNBRIDGE, L.F. MARSHALL and B. RANDELL. "The Newcastle Connection or UNIXes of the World Unite!," *Software-Practice and Experience,* vol. 12, pp. 1147-1162 (1982).

[Devarakonda85]
M. DEVARAKONDA, R. MCGRATH, R. CAMPBELL and W. KUBITZ. "Networking a Large Number of Workstations Using UNIX United," *Proc. 1st IEEE Int. Conf. on Computer Workstations,* pp. 231-239 (1985).

[Feldman78]
S.I. FELDMAN. "Make - A Program for Maintaining Computer Programs," appeared in *UNIX Programmers Manual,* vol. 2A, Bell Laboratories, Murray Hill NJ, 1979.

[Johnson79]
S.C. JOHNSON. "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Sym. on Principles of Programming Languages,* pp. 97-104 (January 1978).

[Miller82]
J.A. MILLER and R.J. LEBLANC. "Distributed Compilation: A Case Study," *Proc. 3th IEEE Int. Conf. on Distributed Computing Systems,* pp. 548-553 (October 1982).

[Renesse84]
R. VAN RENESSE, A.S. TANENBAUM and S.J. MULLENDER. "Connecting UNIX systems Using a Token Ring," Rapport IR-91, Vrije Universiteit, Amsterdam, The Netherlands (October 1984).

[Tanenbaum81]
A.S. TANENBAUM and S.J. MULLENDER. "An Overview of the Amoeba Distributed Operating system," *Operating Systems Review,* vol. 15, no. 3, pp. 51-64 (July 1981).

[Tanenbaum83]
A.S. TANENBAUM, J.M. VAN STAVEREN, E.G. KEIZER and J.W. STEVENSON. "A Practical Toolkit for Making Portable Compilers," *CACM* vol. 26, no. 9, pp. 654-660 (September 1983).

[Tanenbaum84]
A.S. TANENBAUM and S.J. MULLENDER. "The Design of a Capability-Based Distributed Operating System," Rapport nr. IR-88, Vrije Universiteit, Amsterdam, The Netherlands (November 1984).

Parallel Alpha-Beta Search

Henri E. Bal
Robbert van Renesse
Department of Mathematics and Computer Science,
Vrije Universiteit,
Amsterdam, The Netherlands

Several different approaches exist to the design of a parallel alpha-beta algorithm. Recent research in this area is reviewed. Especially, the obstacles to achieving a linear speedup are explained. Alpha-beta algorithms based on tree splitting suffer from search overhead, synchronization overhead, and communication overhead. A new algorithm is developed that avoids the synchronization overhead. This algorithm is implemented under the Amoeba distributed operating system.

1. Introduction

Speed is one of the most important properties of chess programs based on some kind of brute force strategy. High speed can be obtained by using efficient algorithms and fast, possibly special-purpose, hardware. As really fast processors tend to be expensive and usually have to be shared with other people, an interesting alternative is to use *multiple* cheap processors. With today's technology it is quite feasible to build a relatively cheap system with an impressive number of MIPS out of standard hardware.

One problem that has to be solved is how to run a chess program concurrently on many processors. Several researchers have investigated how the *alpha-beta* algorithm (which is the heart of most chess programs) can be run in parallel. In this paper we will give a survey of this research. Especially, we will describe the problems encountered in achieving a linear speedup (i.e., proportional to the number of processors used). Also, the impact of several enhancements to the alpha-beta algorithm on this speedup will be discussed. In the second part of the paper we will report on our experiments with implementing a parallel alpha-beta algorithm under the distributed operating system Amoeba, that has been developed in our faculty [Mullender86, Mullender84, Mullender85].

Parallel Alpha-Beta Search
H.E. Bal and R. van Renesse
Proc. NGI-SION Symposium Stimulerende Informatica
pp. 379-385
Utrecht, Netherlands
April 1986

We assume the reader has a fair knowledge of game theory, in particular of the alpha-beta algorithm and its enhancements. A good historical overview can be found in chapter 2 of [Herik83].

2. A SURVEY OF PARALLEL ALPHA-BETA ALGORITHMS

There are several fundamentally different ways of using parallelism to speed up the alpha-beta search. The static evaluation can be carried out in parallel, for example by having one processor counting the pieces, another one looking for open lines, and so on. Clearly, the speedup will be limited by the number of properties the evaluation function looks at. Furthermore, communication overhead will be substantial, as every processor examines every evaluated position.

Another approach is to do a parallel *aspiration search*. The sequential aspiration search algorithm first tries a small initial window (X, Y). If the search fails, it subsequently tries either (- infinite, X) or (Y, + infinite). A parallel program can try the three windows (- infinite, X), (X, Y), and (Y, + infinite) simultaneously. If there are enough processors, more windows can be used. Baudet [Baudet78] showed that even with an infinite number of processors the speedup is still limited by a factor 5 or 6.

The most promising approach to parallel alpha-beta search is based on *tree splitting*. With this method, each processor searches part of the game tree. In principle, there is no constant upper bound for the speedup, as there is for parallel aspiration search. Yet, there are other problems, that will be discussed below.

2.1. Tree splitting algorithms

Tree splitting algorithms distribute the game tree over all available processors. A basic algorithm works as follows. Initially, one processor is assigned the task of evaluating a game tree. It hands out the leftmost subtree of the root node to subprocessor 1, the second subtree to subprocessor 2, and so on. It computes the solution for the entire tree out of the partial solutions returned by the subprocessors. Each subprocessor can split its own subtree over even more processors. Clearly, this method will soon run out of processors. To avoid this, two precautions are taken. First, each processor has only a limited number of subprocessors, say F. After handing out the Fth subtree, it waits until one of its subprocessors becomes available again. Second, the forwarding of work to subprocessors and to subprocessors of subprocessors is bounded. Essentially, the processors are organized as a *tree* with fan-out F and depth D. After D levels of subcontracting, a processor evaluates the remainder of the subtree itself.

The algorithm has been implemented by Finkel and Fishburn [Finkel82] on a network of 5 LSI-11 processors under the Arachne distributed operating system. Their paper presents theoretical, simulated, and measured results. One problem is that, for a (nearly) best-first ordered game tree, the algorithm evaluates subtrees that would not have been evaluated by a sequential algorithm. For example, during the evaluation of the second subtree of the root node, the

result of the evaluation of the first subtree cannot be used, as these subtrees are evaluated concurrently. Hence, alpha-beta bounds become available much later. As chess programs attempt to achieve a nearly best-first order for their moves, this definitely is a major problem. As a second problem, processors spend part of their time waiting for their subprocessors to finish, hence causing idle-times. Finally, any distributed algorithm will suffer from a certain amount of communication overhead. Because of these problems the speedup obtained by the simple tree splitting algorithm is far from linear. Theoretically, for a best-first ordering the speedup is proportional to the square root of the number of processors.

Summarizing, there are three kinds of overheads: *search overhead*, *synchronization overhead*, and *communication overhead*.

Several researchers have tried to improve this basic tree splitting algorithm. We will discuss two schemes, the Minimal-tree approach (also called the Mandatory-work-first approach) of Akl, Barnard, and Doran [Akl80], and the Principal Variation Splitting method of Marsland and Campbell [Marsland82].

There is a fixed part of the search tree that a (sequential) alpha-beta algorithm cannot possibly cut off. This part is called the *minimal tree*. The mandatory-work-first approach first searches this minimal tree. In a second tree traversal it searches the rest of the tree, using the alpha-beta bounds found during the first traversal. So, during the first traversal the parallel algorithm has no search overhead. During the second traversal the parallel algorithm has less overhead than the simple parallel tree splitting algorithm.

The minimal tree is defined recursively by the following rules:

- the root node is a minimal-tree node
- the leftmost son of a minimal-tree node is itself a minimal-tree node
- the leftmost son of any right son of a minimal-tree node is itself a minimal-tree node.

Finkel and Fishburn [Finkel83] adapted their algorithm to this new approach. Their analysis shows that the expected speedup is significantly better than the speedup of their original algorithm.

Marsland and Campbell [Marsland82, Marsland85, Schaeffer84] proposed an algorithm that does not require two separate tree traversals. Their Principal Variation Splitting (PVsplit) algorithm aims at optimizing searches of *strongly ordered* game trees. PVsplit assumes that most of the time the leftmost subtree of any node will contain the best move. So, PVsplit first evaluates the leftmost subtree and then tries to prove that the other moves are inferior. This is achieved by using a *zero-width* alpha-beta window, causing the search to fail quickly. If the search fails "low," the move is refuted. If a better move is found, the search fails "high" and has to be repeated with a normal window, using the better move as the new principal variation. For strongly ordered trees, the search will fail "low" most of the time.

The algorithm was incorporated in two existing chess programs (TinkerBelle and Phoenix) and implemented on a system of four M68000 based SUN workstations. Experimental results are given in [Schaeffer84]. For deep

searches, a speedup in the range 3.1 - 3.3 is achieved for four processors. The results show that parallelism is most effective for deep searches. For a 3-ply search the search overhead is almost 30%; for a 7-ply search it is about 10% (for four processors).

2.2. The effect of enhancements of the Alpha-Beta search

Sequential chess programs use several enhancements to the basic alpha-beta algorithm, such as transposition tables, refutation tables, history heuristics, aspiration search, iterative deepening, quiescent search, and killer heuristics. In this section we will discuss the effectiveness of some of these enhancements to the parallel alpha-beta algorithms.

A transposition table is a list of moves that have been evaluated earlier during the search. Large transposition tables (containing say 100 000 positions) have proven to be quite valuable. In a parallel environment, a *central* transposition table will soon become a communications bottleneck, as it is consulted often. On the other hand, if each processor maintains its own transposition table, one processor may evaluate moves that another processor has already evaluated. Transposition tables and similar mechanisms were studied by Marsland et al [Marsland85, Schaeffer84]. Local transposition tables and the less expensive refutation tables and history heuristics were found to be superior to one central transposition table.

Moser [Moser84] studied the effect of aspiration search on a parallel alpha-beta algorithm. In his chess program Watchess 3.0, all processors first use a narrow window (X,Y). If this search succeeds, a very good speedup is achieved. (Moser measured a speedup of 5.2 for 7 processors). The main disadvantage of the parallel algorithm (i.e., the delayed availability of good alpha-beta bounds) is compensated for by the tight bounds of the initial search window. The search fails "high" as soon as one processor evaluates a subtree with a value greater or equal than Y. In this case all other processors working on this subtree should be stopped (interrupted) immediately. Although a super-linear speedup (i.e., better than linear) can occur occasionally, in general the speedup is worse than for a successful search. The search fails "low" if no processor detects a subtree with value higher than X. A fairly good speedup is obtained in such a case. Moser concludes that aspiration search is extremely beneficial for distributed alpha-beta searching.

Chess programs always try to order their moves, so they can evaluate plausible moves first. One method, known as iterative deepening, implements an N-ply search by first doing an (N-1)-ply search to order the moves. The (N-1)-ply search first does an (N-2)-ply search. This process continues up to a certain depth. A typical chess program may start with a 2-ply search. If this method is applied to a parallel search algorithm, it will increase the synchronization overhead, as all processors have to wait for the completion of the (N-1)-ply iteration before starting to work on the next iteration [Schaeffer84]. This problem can only be avoided at the cost of a more complex scheduling strategy.

3. The Amoeba distributed operating system

We have implemented the alpha-beta algorithm on top of a modern distributed operating system, *Amoeba* [Mullender86]. In this system there is a minimal kernel per processor capable of nothing more than running processes and providing communication for those processes either locally or over the network. Together they form the bottom layer in the operating system, running communicating processes. The next layer provides services like a file service or a process service. These services complete the operating system by providing the user with the usual mechanisms for reading and writing files, or creating processes. Services can be created dynamically because they are implemented by ordinary processes, called *servers*. Processes that use these services are called *clients*. Of course, a process can be both a server and a client.

Communication between processes is through request-reply pairs: the client sends a request to the server after which it awaits a reply from that server. Thus communication is blocking, as the client is suspended while the server is processing the request. Only when the server is finished and has sent its reply the client can continue to run.

Concurrency is achieved by dividing processes up into sub-processes; in *Amoeba* these are called *clusters* and *tasks* respectively. All tasks of a cluster run on the same processor, so all tasks share their memory. Each task has its own thread of control. A task continues to run until it blocks, and only then another task within the same cluster is allowed to run. This way there is no need for complex synchronization mechanisms to access shared data structures. Each task can start an operation on a separate server, thus enabling concurrent processing as each server can be located on another processor.

In our model there are three classes of processors. A processor can belong to some specific user (i.e., it is part of his personal workstation), to a special server (e.g., a file server), or it can be available for general usage. The latter class of processors form a *pool* that is used to (gradually) enhance the computing capacity of the system. Each user is free to allocate some of these pool processors, for example to run a parallel program.

At present, we have a working prototype of the Amoeba kernel (running on a Motorola 68010, VAX, NS16032, PDP-11, and IBM-PC). This kernel has been used to implement some parallel algorithms, among which is a parallel alpha-beta search algorithm.

4. Parallel Alpha-Beta search on Amoeba

Parallel alpha-beta programs based on tree splitting suffer from three kinds of overheads: search overhead, synchronization overhead, and communication overhead (see section 2). Below, we will develop an algorithm that virtually eliminates the synchronization overhead by always keeping the subprocessors busy.

A parallel alpha-beta search can be implemented on Amoeba as follows (see figure 1):

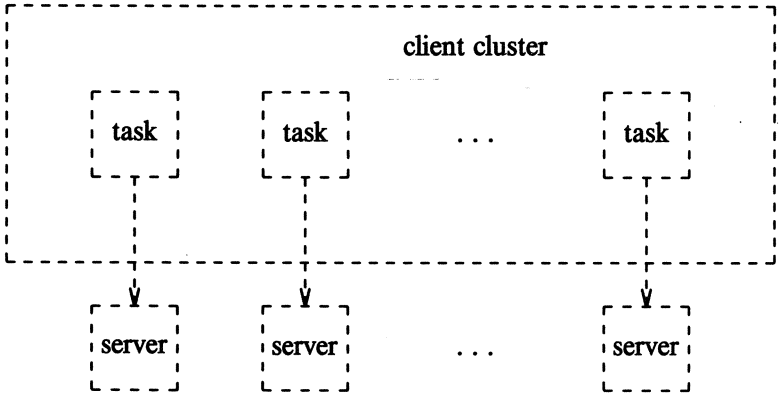


FIGURE 1. Process structure of the Alpha-Beta program.

Start servers capable of evaluating subtrees on each processor, and let one cluster, running on any processor, divide the tree among the servers. This can be done recursively by having each server splitting its subtree once more, and dividing those over yet other servers. Within the cluster each task is waiting for its server to finish; however, because there are as many tasks in a cluster as servers in the system, synchronization overhead is eliminated, so all servers always have some work to do.

Splitting the tree is almost automatic (see figure 2). Each task starts executing the usual sequential alpha-beta algorithm. To keep the other tasks from evaluating the same nodes, each task leaves a trace of what it has done already, or what leaf it is evaluating, by building the tree explicitly in the shared memory. Each task does a depth-first search in the tree until it either finds an unvisited node or an unevaluated leaf, or it decides that the subtree rooted at the current node should be evaluated by another processor. For an unvisited node it will generate all the moves in the corresponding positions and continue in the first child node. An unevaluated leaf is evaluated directly by a static evaluation function. The decision to send a subtree to another processor is based on the current search depth. This also allows efficient alpha-beta interval updates. When a leaf or a node is evaluated, the task that is then executing can update the alpha boundary in the parent node as the node is maintained in shared memory. Furthermore, it can update the beta boundaries in the sibling nodes, and then the alpha boundaries of their child nodes, etc., until the leaves are reached, the new boundary is not better than the old one, or the update results in pruning the rest of the tree.

In the last case the processors that are evaluating parts of that subtree have to be signaled to abort the request, a mechanism that is incorporated in *Amoeba*. When a leaf is reached that is evaluated remotely by doing again an alpha-beta search, and the new alpha or beta is improving the old one, the remote server has to be informed, but only if the communication overhead is smaller than the time to finish the evaluation. Because the mechanism for

doing the remote update would be painful (an asynchronous update in the subtree), and the expected overhead is difficult to estimate, we did not implement this.

```
type node = record      (* definition of a node *)
            position:    position_type;
            alpha, beta: integer;
            busy, remote: boolean;
            children:    list of node;
          end;

  (* Evaluate node.position by doing an alpha-beta search. The result
   * is returned in node.alpha.
   *)
  procedure alpha_beta(node, depth)
  begin
    if depth = 0      (* reached a leaf *)
    then
      node.alpha := static_evaluation(node.position);
    else
      if node should be evaluated by another processor
      then
        node.busy := TRUE;
        node.remote := TRUE;
        send node to server
        wait for server to finish (* here other tasks will run *)
        node.alpha := result;
      else
        if node.children = nil  (* first task to arrive here *)
        then
          generate(node);
          (* generates node.children with alpha-beta boundaries set
           * to (-node.beta, -node.alpha)
           *)
        fi
        foreach child in node.children
        do
          if not child.busy (* no other tasks working on this node? *)
          then
            alpha_beta(child, depth - 1);
          fi
          if child.children = nil  (* all children done *)
          then
            child.busy := TRUE;
            update_alpha(node, -child.alpha);
            (* tells other tasks about result *)
```

```
            dispose of child (* remove child from list *)
          fi
        od
      fi
    fi
  end;

  (* Update node.alpha with alpha.  Update the children too.
   * Remove the node if alpha exceeds beta.
   *)
  procedure update_alpha(node, alpha)
  begin
    if node.alpha < alpha
    then
      node.alpha := alpha;
      foreach child in node.children
      do
        update_beta(child, - alpha);
      od
      if node.children = nil and node.alpha >= node.beta
      then
        if node.remote send abort signal to server
        dispose of node (* node is pruned *)
      fi
    fi
  end;

  (* Update node.beta with beta.  Update the children too.
   * Remove the node if alpha exceeds beta.
   *)
  procedure update_beta(node, beta)
  begin
    if node.beta > beta
    then
      node.beta := beta;
      foreach child in node.children
      do
        update_alpha(child, - beta);
      od
      if node.children = nil and node.alpha >= node.beta
      then
        if node.remote then send abort signal to server
        dispose of node (* node is pruned *)
      fi
    fi
  end;
```

```
begin
  (* initialize root node *)
  node.position := INITIAL_POSITION;
  node.alpha    := -MAXINT;
  node.beta     := MAXINT;
  start N tasks;
  alpha_beta(node, TOTAL_DEPTH);
  wait until all tasks have finished
  writeln('result: ', node.alpha);
end.
```

FIGURE 2. The parallel alpha-beta algorithm.

To measure the performance of our method, we implemented the game of *Othello*. Figure 3 shows the time to evaluate a position, averaged over five different positions with approximately fifteen moves, and the number of evaluations that were not aborted. The *root* cluster searched the tree over three plies; the servers did only one ply after which they did a static evaluation of the resulting position.

version	time(secs)	speedup	#evaluations	search overhead
sequential	266.9		2670	1
1 server	324.6	1	2670	1
2 servers	196.2	1.65	3925	1.47
3 servers	153.3	2.12	4732	1.77
4 servers	125.1	2.59	5676	2.13
5 servers	114.0	2.84	6424	2.40
6 servers	111.5	2.91	6719	2.51

TABLE. Results of an Othello implementation.

5. DISCUSSION

This paper presented a survey of recent research on parallel alpha-beta algorithms. The causes for not achieving a linear speedup were classified as search overhead, synchronization overhead, and communication overhead. An algorithm was presented that virtually eliminates the synchronization overhead, but, in its present form, still suffers from a fairly large search overhead.

Parallelism is used by several existing chess programs, such as Ostrich, Watchess 3.0 [Moser84], and Cray Blitz [Hyatt85]. Most successful is world champion Cray Blitz, that uses a 4 processor Cray X-MP/48. All these programs run on a small number of processors, typically less than 10. With the advent of powerful low-cost microprocessors (like the MC 68020 and the Inmos Transputer) it becomes more and more important to use massive parallelism. Today's parallel alpha-beta algorithms are not well suited for several hundreds of processors, as their speedups strongly degrade. It is still an open research issue whether the existing methods can be improved to make an effective use of many processors.

ACKNOWLEDGEMENTS
The authors would like to thank Jaap van den Herik and Dick Grune for their useful suggestions.

REFERENCES

[Akl80]
AKL, S.G., BARNARD, D.T., and DORAN, R.J., "Design, Analysis, and Implementation of a Parallel Alpha-Beta Algorithm", Report 80-98, Queen's University, Kingston, Canada, April 1980.

[Baudet78]
BAUDET, G.M., "The Design and Analysis of Algorithms for Asynchronous Multiprocessors", CMU-CS-78-116, Carnegie-Mellon University, April 1978.

[Finkel82]
FINKEL, R.A. and FISHBURN, J.P., "Parallelism in Alpha-Beta Search," *Artificial Intelligence*, vol. 19, pp.89-106, 1982.

[Finkel83]
FINKEL, R.A. and FISHBURN, J.P., "Improved Speedup Bounds for Parallel Alpha-Beta search," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. PAMI-5, no. 1, pp.89-92, January 1983.

[Herik83]
HERIK, H.J. VAN DEN, *Computerschaak, schaakwereld en kunstmatige intelligentie*. Academic Service, 1983.

[Hyatt85]
HYATT, R.M., "Parallel Chess on the Cray X-MP/48," *ICCA Journal*, pp.90-99, June 1985.

[Marsland82]
MARSLAND, T.A. and CAMPBELL, M., "Parallel Search of Strongly Ordered Game Trees," *Computing Surveys*, vol. 14, no. 4, pp.533-551, December 1982.

[Marsland85]
MARSLAND, T.A. and POPOWICH, F., "Parallel Game-tree Search", TR 85-1, The University of Alberta, January 1985.

[Moser84]
MOSER, L., *An Experiment in Distributed Game Tree Searching*. University of Waterloo, 1984.

[Mullender84]
MULLENDER, S. J. and TANENBAUM, A. S., "Protection and Resource Control in Distributed Operating Systems," *Computer Networks*, vol. 8, no. 5,6, pp.421-432, 1984.

[Mullender85]
MULLENDER, S. J. and TANENBAUM, A. S., "A Distributed File Service Based on Optimistic Concurrency Control," *Proceedings of the 10th Symposium on Operating Systems Principles*, pp.51-62, December 1985.

[Mullender86]
MULLENDER, S. J. and TANENBAUM, A. S., "The Design of a Capability-Based Distributed Operating System," *The Computer Journal*, vol. 29, no. 4, pp.289-300, 1986.

[Schaeffer84]
SCHAEFFER, J., OLAFSSON, M., and MARSLAND, T.A., "Experiments in Distributed Tree-Search", TR 84-4, The University of Alberta, June 1984.

Experience

Making Distributed Systems Palatable

Andrew S. Tanenbaum
Robbert van Renesse
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

1. Introduction

Designing and implementing a distributed system is easy compared to the task of convincing people to use it. In a university Computer Science Dept., people generally use UNIX† and are not at all interested in moving to a different environment, no matter how wonderful it may be. In this paper we report on how we have implemented a UNIX environment for the Amoeba distributed operating system [1], in order to make the transition from UNIX to Amoeba as simple as possible.

2. Overview of the Amoeba distributed operating system

The Amoeba system runs on a hardware configuration consisting of four components: personal workstations, pool processors, specialized servers (e.g., file servers), and gateways to other systems. Interactive work is done on the workstations. Heavy computing, such as *make*, can be done on the pool processors, which are dynamically assigned as needed. At present there are 16 pool processors, each consisting of a 12.5 MHz 68010 processor and a megabyte of memory. All the components of the system are connected by a 10 mb/s local network.

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

Making Distributed Systems Palatable
Andrew S. Tanenbaum and Robbert van Renesse
Position Paper in 2nd SIGOPS workshop "Making Distributed Systems Work"
Amsterdam, Netherlands
September 1986

The Amoeba software is based on the concept of objects (abstract data types). Each object has some set of operations that can be performed on it A file object, for example, has operations to read it, write it, delete it, etc.

Associated with each object is a capability that gives the holder permission to carry out certain operations on the object. Capabilities contain random numbers, for authentication, and are encrypted, to allow them to be manipulated directly by user processes, without intervention by the operating system.

To perform an operation on an object, a client process sends a message to the server process that owns the object. This request message contains the capability, the operation code, and possibly some parameters. When the server has carried out the operation, it sends a reply message back to the client. The request and reply messages work together to form a simple kind of remote procedure call called a **transaction**. Virtual circuits are not used in Amoeba.

The Amoeba kernel manages the transactions, sending and receiving messages (including splitting long messages into packets), setting timers, handling retransmissions etc. Nearly all of the other traditional operating system functions, such as process management, the file system and even system accounting are done outside the kernel in server processes. This design not only keeps the kernel small, but also makes it possible to have multiple servers of each type, a fact of considerable importance for smoothing the transition from UNIX, as will be described shortly.

One other feature of Amoeba that is worth mentioning is the ability to have multiple processes operating within a single address space. This construction is called a **cluster** of **tasks**. It is particularly useful for implementing multiple threads of control in a server, to allow the server to work on several requests at the same time.

3. THE UNIX SERVERS

Although Amoeba has a number of "native" servers, such as a multiversion file server using optimistic concurrency control, these can coexist with "foreign" servers at the same time, since as far as the operating system is concerned, a server is just another user process. We have taken advantage of this fact, and implemented two servers essential to producing the UNIX environment, a UNIX file server and a UNIX process server.

The UNIX file server, plus an associated library that can be linked in with user programs, provides an interface that is very similar to the UNIX (V7) file system interface. Programs using this library can create and open files, read and write files, and seek on files. Directory operations, including linking and unlinking files, and mounting and unmounting devices are all supported and all work the same way as UNIX programs expect them to work. The net result is that many UNIX programs can be relinked using a special library and run on Amoeba with no modification.

Now let us briefly look at the implementation. The special library contains a procedure for each of the UNIX system calls supported. When a user program wants to execute the READ system call, for example, the library procedure *read* is called. This procedure does a transaction (remote procedure

call) with the UNIX file server, passing as parameters, a capability that effectively establishes its identity (user id and group id), a small integer (file descriptor) telling which file to read, and the number of bytes desired.

When the request message arrives at the server, one of the tasks inside the server accepts it and begins to process it. The server maintains a cache of recently used blocks, so there is a good chance that the data requested will be in the server's memory. If so, the server builds a reply message containing the requested bytes and sends it back to the client. If the data is not in memory, the server task fetch fetches it from the disk. While it is blocked waiting for the disk, requests from other clients can be processed by other tasks within the server.

The UNIX process server has been structured in a similar way. It handles the FORK, EXEC, WAIT, SIGNAL, KILL, and EIXT system calls, among others. When a process forks, it is given a capability identifying the newly created process. The child process uses this capability to identify itself in subsequent operations.

Because nearly all the usual UNIX system calls are supported by one of these two servers, its was straightforward to simply relink many of the standard programs in the UNIX */bin* directory to run under Amoeba. Consequently, users recently moved from UNIX to Amoeba can continue to use the *shell*, various editors, the C compiler, and the small utilities, such as *cat*, *grep*, and *sort*.

4. Communication with UNIX

Amoeba users often want to communicate with UNIX systems, for example, to read their mail. To facilitate this communication, we have written a UNIX driver that implements the standard Amoeba transaction protocol, so that Amoeba processes can communicate with UNIX processes.

An important use of this feature is for implementing the remote shell, *rsh*. Using *rsh*, a person logged into any of the machines, UNIX or Amoeba, can have a command carried out on any other machine. The output of that command is automatically redirected back to the caller's standard output. For example,

```
rsh vax3 who
```

runs the *who* program on a machine called *vax3* and displays the results on the terminal.

Another program, *call* allows an Amoeba user to log into a remote UNIX machine to work their for a while. When the user is done, he logs out and is back to Amoeba.

5. Accessing UNIX files from Amoeba

The native Amoeba file system, FUSS, uses a capability for each file. These capabilities are generally stored in directories, where a directory entry is just an (ASCII string, capability) pair. A user can present a string (path name) to the directory server, and the server returns the corresponding file capability.

We have extended this basic scheme to make it possible to store capabilities

for UNIX files in Amoeba directories in a completely transparent way. On each UNIX machine are two special processes, the **link server** and the **file server**, that make this possible.

To enter a UNIX file into an Amoeba directory, a user does a transaction with the link server, which locates the UNIX file, links it into a special directory of its own, and returns a standard Amoeba capability for it to the caller. This capability can be entered in the Amoeba directory system under any convenient name. Later, when the user wishes to access this file, he asks the Amoeba directory server to look it the name and return the capability. This capability can then be sent to the file server running on the UNIX machine to access the file. This facility has been implemented in such a way that accessing an Amoeba file or accessing a UNIX file are identical from the user's point of view.

When a request to create a capability for a file arrives at the link server, the link server makes a link (in the UNIX sense) to the file, and enters it into an internal directory under a name that is related to the random number in the capability. When the capability is later presented to the UNIX file server for reading, it is possible to check to see if the capability is valid.

The link server can also make capabilities for UNIX directories, although these are implemented differently because the link server cannot link to a directory. Instead an internal table provides the mapping between capabilities and directories.

Reference

MULLENDER, S.J. and TANENBAUM, A.S. *Protection and Resource Control in Distributed Operating Systems.* Computer Networks, vol. 8, Oct. 1984, pp. 421-432.

Making Amoeba Work

Position paper for the 2nd SIGOPS Workshop

"Making Distributed Systems Work"

Sape Mullender
Centre for Mathematics and Computer Science
Amsterdam, The Netherlands

Fifth generation computers must be fast, reliable, and flexible. One way to achieve these goals is to build them out of a small number of basic modules that can be assembled together to realize machines of various sizes. The use of multiple modules can make the machines not only fast, but also achieve a substantial amount of fault tolerance.

The price of processors and memory is decreasing at an incredible rate. Extrapolating from the current trend, it is likely that a single board containing a powerful CPU, a substantial fraction of a megabyte of memory, and a fast network interface will be available for a manufacturing cost of less than $100 in 1990. We therefore do research on the architecture and software of machines built up of a large number of such modules.

In particular, we envision three classes of machines: (1) personal computers consisting of a high-quality bit-map display and a few processor-memory modules; (2) departmental machines consisting of hundreds of such modules; and (3) large mainframes consisting of thousands of them. The primary difference between these machines is the number of modules, rather than the type of the modules. In principle, any of these machines can be gracefully increased in size to improve performance by adding new modules or decreased in size to allow removal and repair of defective modules. The software running on the various machines should be in essence identical. Furthermore, it should be possible to connect different machines together to form even larger

Making Amoeba Work
SAPE J. MULLENDER
Position Paper in 2nd SIGOPS workshop "Making Distributed Systems Work"
Amsterdam, Netherlands
September 1986

machines and to partition existing machines into disjoint pieces when necessary, all in a way transparent to the user level software.

Amoeba [Mullender85a] uses the concept of *objects*, manipulated by *services*. Associated with each object are one or more "capabilities" [Dennis66] which are used to control access to the object, both in terms of who may use the object and what operations he may perform on it. At the user level, the basic system primitive is performing an operation on an object, rather than such things as establishing connections, sending and receiving messages, and closing connections.

The object model is well-known in the programming languages community under the name of "abstract data type" [Liskov74]. When a user process executes one of the visible functions in an abstract data type, the system arranges for the necessary underlying message transport from the user's machine to that of the abstract data type and back. The header of the message can specify which operation is to be performed on which object. This arrangement gives a very clear separation between users and objects, and makes it impossible for a user to inspect the representation of an abstract data type directly by bypassing the functional interface.

The object model is *implemented* in terms of clients (users) who send messages to services [Cheriton83, Needham82, Ball79]. A service is defined by a set of commands and responses. Each service is handled by one or more server processes that accept messages from clients, carry out the required work, and send back replies.

Amoeba has no system calls, apart from the ones for message transactions; process management, terminal handling and accessing device drivers is all done through transactions with services, which may or may not be part of the operating system kernel.

Replicated in each of the Amoeba processors is a copy of the *Amoeba Kernel*, which manages *clusters* of light-weight processes, called *tasks*, and provides communication between tasks through blocking message transactions. Each cluster has a segmented virtual address space that its tasks execute in. Within a cluster, there is no pre-emption between tasks; a task executes until it blocks voluntarily before another taks in the same cluster (and address space) is allowed to run. Task switching can be made very efficient this way.

The Amoeba Kernel manages three types of objects, *clusters*, *tasks*, and *segments*. A cluster consists of one or more tasks wich execute in a single address space, formed by one or more segments. These three types of object can be manipulated by directing requests to the Amoeba Kernel, the *Kernel Server*, so to speak.

The most important data structured used in transactions to manipulate segments, clusters and tasks is the *Cluster Descriptor*, which is capable of describing the complete state—except the *contents* of the segments—of a cluster of tasks.

The Cluster Descriptor plus appropriate Kernel requests, form a single

mechanism for loading and starting up processes over the network, for checkpointing, migration, exception handling and remote debugging. The exception handling mechanisms allow each interaction between a cluster and the outside world to be caught and examined by the exception handler (a user-appointed service), so emulation of other operating system interfaces and encapsulation of Amoeba processes is feasible.

Emulation of other operating system interfaces is important: Amoeba cannot become a popular operating system unless exsiting software can be used on it without, or with very little modification.

The UNIX† system call interface is already available on Amoeba as a special version of the UNIX C-library. For instance, rather than executing a *read* system call upon a call of the *read* routine in C, a transaction is carried out with the file server.

It is planned to use the possibility of encapsulation, mentioned above, to build a UNIX service, which will catch exceptions caused by UNIX binaries doing system calls (or getting stack overflow, or any other exception that may be caused by a UNIX program), and simulate the effect as on a UNIX machine.

Objects in Amoeba are both accessed and addressed through their *capabilities*. A part of the capability, called *port*, for an object specifies the *service* that manages objects of its type. One or more *server* processes (clusters) may be responsible for implementing the service. These processes "listen" on the service's port.

Ports do not carry information about the whereabouts of the associated server processes. The Amoeba Kernels contain a *locate* mechanism to find a server for a service, given the service's port. This mechanism, which is only used in a local network, is based on broadcasting "where are you?" messages and maintaining caches of *hints* on the location of recently-used servers.

For locating ports in wide-area networks which do not normally provide a broadcast facility, mechanisms are needed based on other locate algorithms. We have looked into this problem quite thoroughly and proved a lower bound on the number of message passes needed to locate a port [Mullender85b].

This lower bound indicates that a totally unstructured name space does not scale well, no matter how the network is organised. We are now working on the details of a hierarchical port name space in which a service indicates in what *domain* its servers must expect their clients. Ports still have a fixed length which proves very advantageous for processing speed on local service calls (which are the most frequent), yet the network can be structured hierarchically such that purely local services are invisible in higher levels of the hierarchy.

In collaboration with a dozen European research institutes, partially sponsored by the European Community, we have started work on distributed operating

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

systems for wide-area networks. Amoeba has been chosen as a basis for this work, and research has been done—and is still going on—on methods to connect many local Amoebæ together to form one "Culture" of Amoebæ.

Our special interest in this project is designing operating system services that scale well to very large numbers of processors. Our work on designing algorithms for locating services is one example of this. Another example is protection, authentication and resource control in very large systems.

References

[Ball79]
BALL, J. E., BURKE, E. J., GERTNER, I., LANTZ, K. A., and RASHID, R. F., "Perspectives on Message-Based Distributed Computing," *Proc. IEEE*, 1979.

[Cheriton83]
CHERITON, D. R. and ZWAENEPOEL, W., "The Distributed V Kernel and its Performance for Diskless Workstations," *Proc. Ninth ACM Symp. on Operating Systems Principles*, pp.128-140, October 1983.

[Dennis66]
DENNIS, J. B. and HORN, E. C. VAN, "Programming Semantics for Multiprogrammed Computation," *Comm. ACM*, vol. 9, no. 3, pp.143-155, March 1966.

[Liskov74]
LISKOV, B. and ZILLES, S., "Programming with Abstract Data Types," *SIGPLAN Notices*, vol. 9, pp.50-59, April 1974.

[Mullender85a]
MULLENDER, S. J., *Principles of Distributed Operating System Design.* Amsterdam: SMC, October 1985.

[Mullender85b]
MULLENDER, S. J. and VITANYI, P. M. B., "Distributed Match-Making for Processes in Computer Networks," *Proceedings 4th ACM Principles of Distributed Computing*, August 1985.

[Needham82]
NEEDHAM, R. M. and HERBERT, A. J., *The Cambridge Distributed Computer System.* Reading, Ma.: Addison-Wesley, 1982.

From Unix to a Usable Distributed Operating System

Robbert van Renesse
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Now that we nearly have a production version of the Amoeba distributed operating system ready, we are trying to find ways to make it interesting to UNIX† users. First, we made a full UNIX emulation service for Amoeba. Second, we made the existing UNIX systems fully accessible to Amoeba users, and vice versa. Users can work with Amoeba having their old utilities still accessible, but additionally can use Amoeba services like transparent system-wide file naming and automatic load sharing.

1. Introduction

Distributed operating systems have several well-known advantages, such as

- load balancing;
- network transparency;
- fault tolerance;
- availability.

Trying to get people to use a distributed operating system is hard in spite of these advantages. Reasons:

- not compatible with the old operating system;
- the new operating system is still experimental;
- there are no utilities yet.

For these reasons we want to support UNIX from our distributed operating system, Amoeba. In an attempt to win over the UNIX users, we have put a lot of effort into the utilities and network services. These will be discussed in the following sections. First, however, we will give an overview of the Amoeba

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

From UNIX to a Usable Distributed Operating System
R. van Renesse
Proceedings of the EUUG Autumn '86 Conference
Manchester, UK
pp. 15-21
September 1986

distributed operating system and its communication primitives.

2. Overview of Amoeba

Amoeba [Mullender86] is a distributed operating system being developed at the Vrije Universiteit and the Centre for Mathematics and Computer Science, both in Amsterdam. It currently runs on Motorola 68010, Intel 8086, PDP11, VAX, and National Semiconductor 32016 processors, connected by a Pronet 10 Mbps token ring.

Amoeba is a system of *objects* (abstract data types like files or processes), managed by *services*. Each object is accessed using a *capability* [Tanenbaum86]. Capabilities are cryptographically protected to allow them to be managed by user processes. A service offers a set of operations on its objects, and is made up of a collection of *server* processes. To create, manipulate, or contemplate an object, a *client* process sends a *request* message containing the capability of the object and an operation code to a server process, which will eventually respond with a *reply* message. Such a message exchange is called a *transaction*.

The Amoeba kernel has been kept small by placing services in user space. This enhances its reliability, and allows services to be added, changed, or removed at will, making the operating system flexible. The kernel only provides multi-processing and inter-process communication, both intra-machine and inter-machine. All Amoeba machines run the same kernel.

Inter-process communication in Amoeba is done by *transactions* as mentioned above. Server processes can choose a *port*, an arbitrary 48 bit number, on which they can receive request messages [Mullender84]. A client process can then start a transaction by sending a request message to a port, which is received by a server process that has specified the same port. The transaction is finished when the server has executed the request and sent a reply message back to the client process.

To allow a server process to handle multiple request messages, and a client to do multiple transactions, processes can be divided into lightweight subprocesses. Subprocesses share an address space, and each subprocess is able to send and receive requests. To avoid race conditions and simplify programming, the subprocesses are only rescheduled when a blocking system call is executed, that is, subprocesses are never pre-empted.

Objects in the system are named and protected by *capabilities*. This presents a uniform interface to all types of objects, such as files, processes, directories, devices, disk blocks, etc. An object is created by a service on request of a client, which then receives a capability for it. The capability contains the port of a server in the service, an object number to identify the object within the service, the operation rights associated with the capability, and a check field, redundant information to protect the capability from tampering. Capabilities are managed in user space, again to keep the kernel small.

Capabilities can be stored by the *directory service*[Meer84], which maps ASCII strings to capabilities. It stores the names and capabilities in objects called *directories*, which are named by capabilities too. By storing directory

capabilities in directories, it is possible to build arbitrary naming graphs for capabilities, or, indirectly, for objects.

Besides services, there are three other types of components in the Amoeba system, namely *workstations*, *pool processors*, and *gateways*. Workstations provide an interactive user interface to the Amoeba system. They consist of a kernel running a command interpreter and an editor. Pool processors are dynamically allocated from the *processor pool* when a job has to be run. The gateways [Renesse86] are used to link geographically distributed Amoeba sites into a uniform system.

Ports are located automatically on local Amoeba sites. If a server wants to be known at other Amoeba sites, it *publishes* its port by giving it to the gateway server, together with the *domain* in which the port is to be distributed. Remote gateways will then pass messages intended for the local server over the wide-area network.

3. Transactions

In order to make a transaction, it is necessary to address the server and to name the object to be operated on. The server is addressed by a *port*, a 48 bit number chosen by the server itself. The object is named by a 128 bit number, called a *capability*, which can be subdivided into the port of the server managing the object, an object number that identifies the object itself, the access rights, and a check field to protect against forging capabilities, as shown in figure 1.

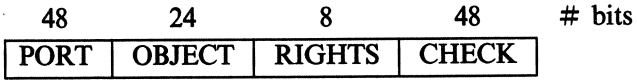


Figure 1. Capability Layout.

To make the use of ports and capabilities clear, consider airline boarding passes, which, as we shall see, are also capabilities. Such a pass contains the name of the flight (the port), the seat you may use (the object number), and the rights you have (*e.g.*, smoking).

Messages consist of two parts: a 32 byte *header* and a *buffer*. The header of a request message contains the capability of the object, a digital signature, an operation code, and some parameters. Only the capability is mandatory. The buffer may contain up to 32 Kbytes of data. A reply message has the same format, although the fields are used differently now. An address need not be specified, as the message is routed automatically to the client that made the request. Rather than an operation code it contains a result code. Figure 2 shows the layout of a header. Byte ordering problems in the header are hidden from the users.

The transaction primitives are listed in figure 3. To await a request message, a server has to specify the header, buffer, and length of the buffer to the kernel by invocating the system call *getreq*. The *getreq* will return the actual size of the received request buffer. To send the reply message back to the client, it

Field	# bytes
Capability	16
Signature	6
Command/Status	2
Parameters	8

FIGURE 2. The *header* format.

calls *putrep*, which will return the length of the reply buffer. When a client wants to send a request message and await the reply, it calls *trans*. The first header and buffer contain the request; the second header and buffer will contain the reply when the transaction is finished. The *trans* returns the actual size of the received reply buffer.

```
getreq(hdr1, buf1, len1)
putrep(hdr2, buf2, len2)
trans(hdr1, buf1, len1, hdr2, buf2, len2)
```

FIGURE 3. The *transaction* primitives.

There are two implementations of these primitives under UNIX. The first one uses Berkeley sockets. The other implementation is a driver for UNIX, which is running under BSD 4.1, BSD 2.9, V7, System III, and System V flavors of UNIX systems. This implementation is compatible with the Amoeba network, and makes communication between UNIX systems and Amoeba systems possible. Furthermore, it supports the transaction primitives in the UNIX kernel itself, so that remote disks and remote terminals can be implemented.

4. UNIX SERVICES

Now that we have a uniform communication interface for Amoeba and UNIX, we can make any UNIX system available to Amoeba users and users on other UNIX systems, by running servers that give access to local resources such as files and processes. This section discusses some of these services.

The *rsh* service provides remote execution and file transfer to its clients, that is, it can start a remote UNIX process and connect input and output streams to the local site. Each UNIX site runs an rsh server. When this server gets a request, in the form of a shell command, it starts the shell and connects its input and output to pipes. On the other ends of the pipes are processes that transfer data between the client and the running shell command.

The UNIX *rsh* command invokes a remote rsh server. The syntax is

```
rsh [—i] machine [ command [ args ... ] ]
```

This runs the command at the specified machine. Input is only read and transmitted to the remote UNIX site if the —i flag is given. If no command is given, an interactive shell is assumed. For example:

```
rsh machine who
```

shows who is logged on to the named UNIX machine. To transfer a file from

the local machine to another, one could do:

```
rsh —i machine "cat > file" < file
```

Using rsh, several shell scripts have been written to implement *rwho*, *rcat*, *rcp*, and others. Rsh is also used to transfer mail between UNIX sites.

More transparent remote execution and file transfer can be achieved by relinking all the UNIX software with a library package we have implemented, with procedures that are used instead of the UNIX system calls. File names are parsed to see if they are of the form *machine!file*. If this is the case the system calls associated with such a file name are executed on the specified machine, instead of on the local machine. Now it is possible to use standard system utilities, but with a global name space. For example:

```
mach1!cat mach2!file
```

prints the file located at the UNIX site *mach2*, running the *cat* command at the machine called *mach1*.

A similar library package, optimized for its purpose, was implemented to make a version of *rn*, that reads the USENET news, using a news spool directory on a remote UNIX system. This was done to avoid having copies of this spool directory on all the UNIX workstations in our department.

To make UNIX files accessible to Amoeba users in the same way as any Amoeba object, UNIX files need to be named and accessed using capabilities. For this purpose we have implemented a UNIX file service that creates capabilities for UNIX files on request and allows holders of these capabilities to read and write the associated files [Storm85]. Access rights are maintained using the rights bits in the capabilities.

5. AMOEBA SERVICES

Having capabilities for UNIX files, it is possible to store them in the Amoeba directory service. Then the UNIX files are linked in the Amoeba directory structure together with Amoeba files and other Amoeba objects. The locations of the UNIX files are invisible, making it possible to create a transparent naming space. Files and directories are accessible through transactions from any UNIX machine or from Amoeba.

Another Amoeba service that is useful for UNIX users is the *terminal concentrator*. This is an Amoeba processor with several terminals attached to it, that can be read or written with transactions. By installing a special character device driver in a UNIX system, the terminals become accessible to that system. The terminal concentrator supports several line disciplines, so that usually a transaction is done per line of input rather than per character, reducing network and operating system overhead. Also, before starting a session, the terminal concentrator asks the user to which system it wants to be connected, giving a flexible terminal configuration.

Remote login from one UNIX system to another is enabled by a UNIX program that simulates an Amoeba terminal concentrator that has one terminal attached to it. It is invoked by

call machine

Other Amoeba devices than terminals can be used by UNIX systems in the same way. For example, a UNIX driver that reads and writes by doing transactions with the Amoeba disk service has been written, implementing a remote disk. This disk can be shared by more than one UNIX system if it is mounted read-only on all the systems. Using the Amoeba boot service it is possible to download machines with a UNIX kernel. These machines may well be diskless.

6. MINIX

Minix is an Amoeba service that implements a UNIX V7 service. It is divided into two servers: a file server and a process server. The file server implements the UNIX *creat*, *open*, *read*, *write*, *close*, *dup*, and the other system calls that operate on files. The process server supports *fork*, *exec*, *exit*, *wait*, *signal*, *kill*, *getpid*, *getuid*, etc.

The *Minix* service is invoked using so-called *stub-routines*, procedures that hide the transaction details from the caller, thus implementing *remote procedure calls*[Birrell84]. Here the stub-routines' syntax and semantics are identical to those defined by the seventh edition of the UNIX Programmer's Manual. The processes authenticate themselves to the *Minix* servers by presenting a capability. The object number in the capability is in effect the UNIX process identifier.

The file server is implemented by an Amoeba server process. It uses a file system structure similar to that of the UNIX V7 system, with some added improvements. The file server uses the Amoeba disk service for storage, and keeps the disk blocks in a large cache. The terminal concentrator is used for terminal access.

The process server uses the file server to read executable files, or create and write core files on process crashes. When it gets a request to *exec* a file, it allocates a processor from the *processor pool*, and copies the file from the file server to a memory segment in this processor. Memory segments behave just as ordinary Amoeba files. Furthermore, it creates a stack segment at this processor and a process with the two segments mapped in the process's address space.

The *Minix* service enables Amoeba users to run UNIX programs as if they were running under UNIX; in effect, Amoeba has inherited an enormous range of application software from UNIX.

7. PERFORMANCE

In this section we present some performance figures for transactions. We measured transaction transfer rate and response time (*i.e.*, the time between sending a request and receiving a reply) for different sizes of the request buffer. Here the server does nothing but accept the requests and send null replies back immediately.

Transaction overhead includes server location time, DMA time (and possibly

buffer copy time), network transmission time, interrupt latencies, context switch times, and system call overhead. We measured this for two 12.5 MHz 68000 processors running UNIX System V.0, and two 10 MHz 68010 processors running Amoeba, both pairs communicating over a Pronet 10 Mbit token ring. The results can be seen in figure 4.

UNIX to UNIX		
buffer size (bytes)	transfer rate (Kbytes/sec)	response time (msec)
0	0	11
1024	79	13
2048	79	26
30000	150	200

Amoeba to Amoeba		
buffer size (bytes)	transfer rate (Kbytes/sec)	response time (msec)
0	0	6
1024	92	11
2048	109	19
30000	249	120

FIGURE 4. Performance figures.

Amoeba performs better, since the transactions are an integral part of the Amoeba operating system, whereas transactions in UNIX are implemented by a driver. Therefore system call overhead is larger in UNIX, and there is also extra copy time since it is not possible (for portability reasons) to DMA directly to user space as is done in Amoeba (note that this has to be done on both client and server side). Since the packet size on Pronet is 2044 bytes, there is not much difference in transfer rate for 1024 byte and 2048 byte buffers.

8. CONCLUSIONS

Supporting the Amoeba transaction interface under UNIX has made communication between the two operating systems possible, and thereby allowing exchange of services. This will help make Amoeba attractive to UNIX users. The transaction driver for UNIX is portable to many different UNIX flavors, and can be installed both on big mini-computers and on small UNIX workstations.

Because the transaction interface is easy to use, simple communicating processes can be implemented. For example, *rsh* makes remote execution on UNIX systems possible. In spite of a simple request-reply interface, data transfer is as fast or faster than if a virtual circuit interface had been used.

UNIX applications are supported on the Amoeba distributed operating system using a special service that implements all the UNIX system calls. In fact, the service makes it possible to build a distributed UNIX system. In our system, however, the service is only used to extend the services offered by Amoeba itself.

REFERENCES

[Birrell84]
BIRRELL, A. D. and NELSON, B. J., "Implementing Remote Procedure Calls," *ACM Trans. Comp. Syst.*, vol. 2, pp.39-59, Februari 1984.

[Meer84]
MEER, THEO J. VAN DER and WELMAN, CARL G. M., "A Capability Server for the Amoeba Distributed Operating System", Master Thesis, Vrije Universiteit, Amsterdam, April 1984.

[Mullender84]
MULLENDER, S. J. and TANENBAUM, A. S., "Protection and Resource Control in Distributed Operating Systems," *Computer Networks*, vol. 8, no. 5,6, pp.421-432, 1984.

[Mullender86]
MULLENDER, S. J. and TANENBAUM, A. S., "The Design of a Capability-Based Distributed Operating System," *The Computer Journal*, vol. 29, no. 4, pp.289-300, 1986.

[Renesse86]
RENESSE, R. VAN and STAVEREN, J. M. VAN, "Wide-Area Communication under Amoeba", IR-117, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December 1986.

[Storm85]
STORM, T. W. VAN DER, "Link Server and File Server, two Amoeba style Servers on UNIX", Master Thesis, Vrije Universiteit, Amsterdam, July 1985.

[Tanenbaum86]
TANENBAUM, A. S., MULLENDER, S. J., and RENESSE, R. VAN, "Using Sparse Capabilities in a Distributed Operating System," *Proc. of the 6th Int. Conf. on Distributed Computing Systems*, pp.558-563, May 1986, Vrije Universiteit.

Accommodating Heterogenety in the Amoeba Distributed System

Sape J. Mullender
Centre for Mathematics and Computer Science
Amsterdam, The Netherlands

Robbert van Renesse
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

1. Introduction

Amoeba [Mullender86] is a general-purpose distributed system, designed and implemented by co-operating groups at the *Vrije Universiteit* and the *Centre for Mathematics and Computer Science,* both in Amsterdam. The system can accommodate four types of machines: (1) powerful personal workstations, that may contain several processors, (2) machines dedicated to providing specific services, such as a distributed file service, printer service, etc., (3) a *processor pool*, consisting of a number of processor/memory pairs, which provides temporary computing power for the users of the system, and (4) guest computers, running other operating systems than the Amoeba Kernel, which may provide services to Amoeba or use services provided by Amoeba.

From the outset, Amoeba was designed with heterogeneity in mind. The remainder of this paper briefly describes how it is accommodated.

2. Efficiency and portability in communication

Efficiency and portability are often mutually exclusive properties of an interprocess communication mechanism: Portability means independence of chosen CPU-type, network type, and network interface. This often means that hardware features cannot be fully exploited.

The Amoeba system uses a request/reply communication protocol: a client

Accommodating Heterogeneity in the Amoeba Distributed System
Sape J. Mullender and Robbert van Renesse
Proceedings of SOSP Heterogeneity Workshop
Orcas Island, Washington, USA
December 1985

process sends a request to a server process. The server executes the request and returns a reply. The reply serves as an acknowledgement for the request, although a separate acknowledgement is sent if processing a request takes a long time. A new request serves as acknowledgement for the previous reply (but an acknowledgement is sent if no new request is forthcoming). It is a stop-and-wait protocol. Requests and replies can be up to 32 Kbytes long, and may have to be fragmented for transmission [Mullender84].

When this protocol was first implemented, it became clear that the machine-, operating system- and network-dependent part of the code far exceeded the portable part, the protocol itself. It also appeared that different networks react very differently to slight variations between protocols. On our token ring, for instance, the first implementation sent all fragments of a message as fast as the network would carry them; this failed, because the receiver interface was singly buffered and had to put received packets in memory before it was ready again. We decided to use a stop-and-wait protocol for fragment transmission. This was improved by making use of the fact that the interface allows independent operation of transmitter and receiver; an acknowledgement can thus be prepared while the receiver is working. The result is an implementation that can transmit over 300 Kbytes per second between the user spaces of processes on different machines (10 Mbit ring and 1 μsec per byte DMA).

Going to these lengths when implementing IPC for a distributed operating system is justifiable, because all traditional operating system services are nowat least potentiallyremote, and the performance penalty for accessing them must be kept as low as possible. We therefore decided to standardise the interfaces to our protocols and their semantics, but not their implementation: each implementation must allow exploitation of the possibilities offered by the network type and interface.

We have given two reasons for doing this: implementation-dependent code far exceeds the code to implement a simple protocol and (2) the penalty for inefficiencies in interprocess communication throughput and response time are high; they make the difference between a usable and an unusable system.

3. PROTECTION

Usually, protection mechanisms are enforced with the help of a secure operating system. In distributed systems this is not possible, because participating guest operating systems may not enforce the right kind of protection, and it is too easy to replace the secure operating system in one of the machines in the system by another, insecure one.

In Amoeba, protection is provided for by the interprocess communication mechanisms [Mullender85]. A process can only send a message to another if it has a **capability** for it. These capabilities consist of bit patterns; knowledge of the pattern is needed to obtain access to the object it refers to. Capabilities can be kept in user space. Secrecy is the key to protection in Amoeba.

For reception of messages a similar capability is required. It is not the same as the one for sending, however, to prevent a sender from impersonating the receiver as well. The capability for sending is the **put-port**, that for receiving

the **get-port**.

Two implementations are possible, one using public-key cryptography (sending requires the encryption key, receiving the decryption key; the keys are used as capabilities), the other using one-way functions. We shall briefly describe the latter.

Get-ports and put-ports are related through a one-way function, a function whose value is easy to compute, but, given the result of such a computation, it is infeasible to compute the input. Thus,

$$put\text{-}port = F(get\text{-}port).$$

each machine and the network, we assume, exists a small box (either physically, or conceptually, as discussed below). This box, the **F-box**, performs the following functions: When a *receive* operation is done by the host, it passes the *get-port* to the F-box. The F-box computes the *put-port*, using *F*, which is publicly known, by the way, and waits for messages with that *put-port* in the header.* The sender, when it sends a message, addresses it with the *put-port*; a *get-port* for the return message traffic is also included. The sender's *F-box* does not operate on the *put-port*, but it does convert the *get-port* to a *put-port*.

If the one-way function cannot be broken and the *F-box* cannot be circumvented, this method provides a protection mechanism that is independent of the security of the operating system. Ideally, the F-box would be built in hardware, out of reach of malicious users, e.g., in the cable ducts in the wall with some sort of alarm that goes off when the *F-box* is tampered with. But the *F-box* can also be put in the VLSI interface chip, on the interface board, or, if need be, in the operating system kernel, where it is as secure as any distributed operating system.

4. Guest systems

No considerations for existing (centralised) operating systems were taken into account in the design of the Amoeba system. In spite of this decision, integrating software from other systems and communication with other systems has not presented a problem. The simplicity of the Amoeba model has made this possible.

Communication and portability between our UNIX† systems and the Amoeba system has been realised by putting an Amoeba *driver* in the UNIX kernel, and a UNIX interface in one of the Amoeba run-time libraries. With these facilities we can test Amoeba software on the UNIX systems, we can build services that run under UNIX and provide access to UNIX services from the Amoeba system, and we can run UNIX software on Amoeba [Renesse84].

The Amoeba driver effectively provides UNIX processes with extra system calls for sending and receiving Amoeba requests and replies, both locally and remotely. Services can thus be set up to provide access to facilities offered by

* For simplicity, we assume a broadcast network

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

UNIX, such as the file system.

The UNIX interface for Amoeba consists of a library that is linked into UNIX programs. The library interprets system calls, such as *open*, *read*, and *write* by sending appropriate requests to a UNIX-like file server, called Monix. Even system calls, such as *fork* and *exec* could be interpreted with little trouble. Many programs, written for UNIX, now run on Amoeba without any significant performance penalty.

This approach to interfacing to guest operating systems has been most useful: No concessions were needed in the design of Amoeba, the interfaces (driver plus library) were written and tested in weeks rather than months, and the UNIX population is happy with some new distributed services.

References

[Mullender84]
MULLENDER, S. J., "A Secure High-Speed Transaction Protocol", Report CS-R8417, Centre for Mathematics & Computer Science (CWI), Amsterdam, October 1984.

[Mullender85]
MULLENDER, SAPE J. and TANENBAUM, ANDREW S., "Protection and Resource Control in Distributed Operating Systems", Report IR-79, Vrije Universiteit, Amsterdam, June 1985.

[Mullender86]
MULLENDER, S. J. and TANENBAUM, A. S., "The Design of a Capability-Based Distributed Operating System," *The Computer Journal*, vol. 29, no. 4, pp.289-300, 1986.

[Renesse84]
RENESSE, ROBBERT VAN, TANENBAUM, ANDREW S., and MULLENDER, SAPE J., "Connecting UNIX Systems Using a Token Ring," *Proceedings of the Cambridge EUUG Conference*, September 1984.

Connecting Unix Systems Using a Token Ring

Robbert van Renesse
Andrew S. Tanenbaum
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

Sape J. Mullender
Centre for Mathematics and Computer Science
Amsterdam, The Netherlands

As part of the research on distributed operating systems being done at the Vrije Universiteit, we have implemented a set of network-oriented programs for use on several UNIX† machines connected by a high-speed token ring. With these tools it is possible to transfer files between machines, log in to remote machines, and implement multimachine shell scripts. The transaction protocols discussed in another paper at this EUUG meeting are used to implement two basic services: a "shell server" and a data transfer service. Other services are easily implemented as shell scripts that use these services. A file transfer program, for instance, executes the command "to < file1" on one machine, and "from > file2" on the other machine. More examples of these facilities and their implementation and performance are discussed in the paper.

1. Introduction

At our university we are developing a distributed operating system called Amoeba [Mullender86]. As a spin off from this research, we have incorporated some of the Amoeba interfaces into UNIX, and used these interfaces to build some application programs for communicating between UNIX systems. These tools include file transfer, remote execution and remote login. In this paper we describe the different layers into which our implementation is divided, and the interfaces that connect them, and discuss the performance of our implementation.

When we started this project we had 2 PDP11/44's running UNIX V7, 2 VAX 750's* running Berkeley 4.1BSD and 8 Intel 8086's and 8 Motorola 68000's running Amoeba 1.0. As Amoeba was designed to be a distributed

† UNIX is a Trademark of AT&T Bell Laboratories.
* PDP, VAX and UNIBUS are registered trademarks of Digital Equipment Corporation.

Connecting UNIX Systems Using a Token Ring
Robbert van Renesse, Andrew S. Tanenbaum, and Sape J. Mullender
Proceedings of the Cambridge EUUG Conference
September 1984

system, we needed a network.

Our network had to be fast, even under heavy load, so a ring network seemed the best choice. After some study, we chose ProNET.*
[Saltzer80] This is a 10 Mbit/sec star shaped ring network with decentralized control and token arbitration, supporting up to 255 hosts. It can send and receive packets concurrently, do scatter/gather operations, has variable length packets up to 2044 bytes, checks parity, and has a primitive hardware acknowledgement bit. Pronet interfaces exist for UNIBUS and MULTIBUS†; both are used in our machines.

Our desires, with respect to UNIX, were modest. We did not want to make a distributed system, but only some capabilities to do file transfer and remote execution. In retrospect, we feel that we have achieved these objectives.

2. Network interface

Network application programs need a mechanism to commmunicate reliably. We have designed a network interface that is simple to use, which uses a efficient, simple and fast protocol. We envision communication between two processes, one is called the *server* and the other the *client*. A server handles requests from clients. When the server has handled the request it sends a reply back to the client; the sending of a request to the server and a reply back to the client is called a *transaction*‡ [Mullender84].

The transaction primitives are:

```
typedef struct Mref {
        char    *M_oob;
        char    *M_buf;
        unsigned M_len;
} Mref;
```

The client, in order to do a transaction calls

```
trans(cap, req, rep);
        Cap *cap; Mref *req, *rep;
```

The server receives requests and sends replies with

```
getreq(port, cap, req);
        Port *port; Cap *cap; Mref *req;

putrep(rep);
        Mref *rep;
```

* ProNET is a trademark of Proteon Associates, Inc.
† MULTIBUS is a trademark of Intel, Inc.
‡ Not to be confused with the concept "atomic transaction."

3. User programs

And now the moment of truth: can the primitives we designed be used to make useful programs? The basic things we want are file transfer and remote execution. In this section we will discuss some of the programs we have built; they fulfilled our desires and are now among the most-used programs on our UNIX systems.

3.1. File transfer

The first thing expected of a fast local network is fast file transfer. We have made two simple programs to accomplish basic data transfer, requiring the user to be logged in on both the machine producing the data, and the machine consuming the data. Their syntax is:

from identifier
to identifier

To reads from standard input and *from* writes to standard output. If the identifiers of *to* and *from* are the same, the input data to *to* becomes the output data of *from.*

For example, when "to hamlet < /etc/passwd" is executed on machine A, and "from hamlet > /etc/passwd" on B, the password file of machine A is copied to the password file of B. The same can be done with the execution of "rcp A!/etc/passwd B!/etc/passwd," called at any machine on the network; *rcp* will be treated in a later section.

3.2. Remote login

As programmers are lazy, they do not like to walk from terminal to terminal to work on different machines, especially if the terminals are in different rooms, floors or buildings. So a desire existed to be able to login onto any computer from any terminal; therefore, we made our own version of the *cu* command to call another UNIX system, except that our version does not lose characters. The syntax is:

call machine-name

After calling this program you get a login message from the remote machine, and you can login and work onto that machine as if the terminal is connected directly to the new machine, with one exception: lines beginning with a 'T' are special. Their meaning is as follows:

T.: switch back to local machine;
T!: shell escape;
TT: send a 'T' to the remote machine;
T%take from [to]: copy file "from" to local machine;
T%put from [to]: copy file "from" to remote machine.

To execute *call* you will have to be logged in on some machine. If you are not, you can login as "remote." Instead of a shell you get a program that asks you for the machine you want to login on, and then executes *call.*

So each terminal is effectively connected to each machine. At the moment, if you inspect what each user is doing in our department, you will notice that half of them are executing *call*. It is useful because most of our machines are dedicated to one or two specific projects, and most of the faculty members are working on projects on different machines.

3.3. Remote execution

Many times you just want to execute a simple command at a remote machine without going to the trouble of logging in; *e.g.*, you want to know if you are still in the top 10 of your favourite game on a certain machine, and if you are not you will have to login on this machine to fight for your place. Commands are executed on a remote machine with:

```
rsh machine command
```

The output of the command is defaulted to the user's terminal, but can be redirected in the usual way, the input comes from "/dev/null." For example, "rsh A who" will give you a listing of the person's who are logged in on machine A.

It is now possible to run your programs on multiple machines. For example, if you want to run an *neqn/nroff* job, you could run it on two machines as follows:

```
(neqn file | to format)&
rsh machine "from format | nroff -ms" > out
```

The *nroff* output is redirected to the file "out" on the local machine. If you want to direct input to the remote command, and split standard output and error output, you could do something like this:

```
rsh machine "from input | command | to output" >&2 &
from output&
to input
```

This means: execute *command* at the remote machine, with input from the process "from input" and output to "to output." Locally a "from output" is started in the background to catch the standard output of the command; the standard input is sent to the remote machine with "to input." The error output is done by the *rsh* process. If you put all this in a shell script, you can execute a command as if it runs locally. In the special case that this command is "sh -i," you can almost work on the remote machine as if logged in there.

3.4. Other useful programs

Out of the basic elements of file transfer and remote execution many interesting programs can be built. In this section we will discuss the programs used most on our machines; all these programs are shell scripts. Many of these scripts call *to* and *from*, which need a unique identifier as argument; for this purpose, the program *uniqport* outputs a random string of printable characters, to used as argument to *from* or *to*. The presented implementations of the

programs are slightly simplified.

For file transfer it is a nuisance to have to login on two machines; therefore, we made a shell script called *rcp* which transfers files from any place in the network to any other place. Its syntax is:

```
rcp [machine1!]file1 [machine2!]file2
```

This will transfer the first file to the second. One can leave out the machine part if the file is on the local machine. An implementation, in which the machine parts are non-optional, could be:

```
IFS=! port=`uniqport`
(set $1; rsh $1 "cat $2 | to $port")&
(set $2; rsh $1 "from $port | cat > $2")
```

A program related to *rcp* is *rcat*, with syntax:

```
rcat [-] [machine!]file ...
```

and obvious meaning.

An implementation of this command, with exactly one file argument, could be:

```
IFS=!
set $1
case $# in
1) cat $1 ;;
2) rsh $1 cat $2 ;;
*) echo "Usage: $0 [machine!]file" >&2 ;;
esac
```

Here is another thing about programmers: they are nosy. They want to know where their fellow-programmers are logged onto, and what they are doing. For this purpose we created the programs *rwho* and *rw*, which give information about the whereabouts and actions of all person logged in on any machine.

In our department we have several different printers attached to several different machines. Some produce ugly output fast, others produce pretty output slowly. It would be nice to print a file on an appropriate printer, independent of the machine the printer is attached to, or the system the file is on. With the program *rpr* you can do the same as with *lpr*, but with the advantages of location independence:

```
rpr printer [file ...]
```

Its implementation, in a configuration having two printers on the machine called "tjalk" and one on the machine "klipper," is:

```
case $printer in
tjalk)   mac=tjalk   com=lpr ;;
pmds)    mac=tjalk   com=opr ;;
klipper) mac=klipper com=lpr ;;
*)       echo "$0: unknown printer" >&2; exit 1 ;;
esac
port=`uniqport`
rsh $mac "from $port | $com" &
shift
pr -t $: | to $port ;;
```

Each shell script was written in 1 to 15 minutes; the basic elements of our network utilities (*from*, *to* and *rsh*) have proved their strength.

3.5. Implementation

Now having described the communication programs and the shell scripts we have built with them, we will discuss how *from*, *to*, *rsh* and *call* are implemented; in particular, we will take a look at the servers needed. All these programs use transactions as communication mechanism.

The implementation of *from* and *to* is simple: *from* acts as a server waiting for request to output data to standard output, *to* acts as a client doing transactions requesting the *from* process to output the data *to* has read. The port used in the transaction header is just the identifier given as argument to *to* and *from*.

To execute a command on a remote machine, a server is needed that awaits a request and executes it when one arrives. The *rsh* command is nothing but a client process doing a transaction with this server, requesting a command to be executed, and awaiting a reply saying the command has been executed. The servers on the different machines listen to different ports; given a machine's name, *rsh* knows the port to use*.

For remote login one also needs a server. Although the server for remote execution could be used for this purpose too, a new one is made. A simple-minded implementation of *call* could be the following:

```
rsh machine "from input | sh -i" &
to input
```

The problem here is that the remote shell has pipes for input and output; for example, you can not do ioctl's, or send signals along pipes. Therefore, we installed a device driver implementing a "pseudo terminal." The job of the remote login server is to manage these pseudo terminals.

A pseudo terminal really consists of two devices: a master and a slave device. The master device can be opened by a process simulating the terminal by writing to it for terminal input, or reading from it for terminal output; the

* The port is a function of the machine's name.

slave device just looks like a terminal device to UNIX. The master device is called "/dev/ptyXX," and the slave device "/dev/ttyXX." The slave device is put in "/etc/ttys" as the other terminals are, so a *getty* process can manage it. The master device has two processes driving it: the first writing to it simulating the pseudo-keyboard, and the second reading from it simulating the pseudo-printer. These processes are just *from* and *to*, so that the pseudo terminal can be controlled at the local machine. All the remote login server does when it gets a request, is pick a free pseudo terminal and start the *from* and *to* processes.

The client process *call* sends a message to the server requesting for a pseudo terminal, sets the local terminal in RAW mode, and starts a *from* and a *to*. The *from* catches the output from the pseudo terminal, and the *to* will send its input to the pseudo terminal. *Call* just copies its input to the *to* process via a pipe, except for the lines beginning with a '~', for which it must do some local processing.

As an example of how this mechanism works, we will consider what happens when the user types a DEL character, with the intention to generate an interrupt at the remote machine. First, the DEL is read by the local terminal driver, but because it is working in RAW mode, it just passes the character to the reader: the *call* process. *Call* outputs it in the pipe, giving the DEL to the *to* process, which sends it to the remote *from* process; *from* writes it to the controlling site of the pseudo terminal device. Now the DEL character is treated as if the pseudo terminal was an ordinary terminal where a DEL was typed in: an interrupt is sent to all the processes belonging to the process group of this terminal.

Although the characters typed in when executing *call* pass through a pipe, are sent to and echoed by the remote machine, and thus sent over the network twice, they are sent back to the terminal fast enough to see only a delay in the exceptional case of a lost packet, when the corresponding character has to be retransmitted. All the network programs are fast enough to work with, even by impatient programmers; but their success is mostly because of the simplicity of usage.

4. Performance

In this section we will give some performance figures for the rates we achieve using *from* and *to*. They were measured during the middle of the day, *i.e.*, many persons were logged in, of whom some were working. Running the tests on a single user system sometimes doubles the data rate, but these figures are not of any importance, since in practice the systems are always multiuser. On the other hand, the performance drops fast if the systems are heavily used. The rates, as shown in figure 1, are not bad compared to most other systems.

	VAX 750	PDP 11/44
VAX 750	25,000	15,000
PDP 11/44	15,000	10,000

FIGURE 1. Data transfer rates in bytes per second over ProNET from user process to user process. The VAX's run 4.1BSD, and the PDP's V7. The buffer size is 512 bytes.

When we made the buffer size 2048 bytes on the VAX's, we achieved a data rate of 90,000 bytes per second (without file I/0). Unfortunately we could not use this size in general as we could not enlarge the buffer size on the PDP's.

As it does not matter where you run the network software, you may also run *from* and *to* on the same machine. The rates we achieve now are in figure 2. As these rates are the same as when running *from* and *to* locally, we may conclude that ProNET is not the bottleneck, but either the protocol or UNIX. Since our protocol is light weight, it must be UNIX. Indeed, when we look at where the most time is spent, it is in copying the user buffer to a kernel buffer, and in setting the timers.

VAX 750	PDP 11/44
25,000	10,000

FIGURE 2. Local rates. *From* and *to* both run on the same machine, and do not use ProNET.

REFERENCES

[Mullender84]
MULLENDER, S. J. and RENESSE, R. VAN, "A Secure High-Speed Transaction Protocol," *Proceedings of the Cambridge EUUG Conference*, September 1984.

[Mullender86]
MULLENDER, S. J. and TANENBAUM, A. S., "The Design of a Capability-Based Distributed Operating System," *The Computer Journal*, vol. 29, no. 4, pp.289-300, 1986.

[Saltzer80]
SALTZER, J. H. and POGRAN, K. T., "A Star-Shaped Ring Network with High Maintainability," *Computer Networks*, no. 4, pp.239-244, 1980.

Contributing Authors

Erik H. Baalbergen [2]
Henri E. Bal [2]
Jane Hall [3]
Sape J. Mullender [1]
Robbert van Renesse [2]
Hans van Staveren [2]
Andrew S. Tanenbaum [2]
Paul M.B. Vitányi [1]

1 Centre for Mathematics and Computer Science
Kruislaan 413, 1098 SJ Amsterdam, Netherlands
phone +31 20 592 9333, telex 12571 MACTR NL

2 Subfaculteit Wiskunde en Informatica, Vrije Universiteit
De Boelelaan 1081, 1081 HV Amsterdam, Netherlands
phone +31 20 548 8080

3 Computer Science Division, Hatfield Polytechnic
P.O. Box 109, College Lane, Hatfield, Herts AL10 9AB, United Kingdom
phone +44 7072 79345, telex 262413

CWI TRACTS

1 D.H.J. Epema. *Surfaces with canonical hyperplane sections.* 1984.

2 J.J. Dijkstra. *Fake topological Hilbert spaces and characterizations of dimension in terms of negligibility.* 1984.

3 A.J. van der Schaft. *System theoretic descriptions of physical systems.* 1984.

4 J. Koene. *Minimal cost flow in processing networks, a primal approach.* 1984.

5 B. Hoogenboom. *Intertwining functions on compact Lie groups.* 1984.

6 A.P.W. Böhm. *Dataflow computation.* 1984.

7 A. Blokhuis. *Few-distance sets.* 1984.

8 M.H. van Hoorn. *Algorithms and approximations for queueing systems.* 1984.

9 C.P.J. Koymans. *Models of the lambda calculus.* 1984.

10 C.G. van der Laan, N.M. Temme. *Calculation of special functions: the gamma function, the exponential integrals and error-like functions.* 1984.

11 N.M. van Dijk. *Controlled Markov processes; time-discretization.* 1984.

12 W.H. Hundsdorfer. *The numerical solution of nonlinear stiff initial value problems: an analysis of one step methods.* 1985.

13 D. Grune. *On the design of ALEPH.* 1985.

14 J.G.F. Thiemann. *Analytic spaces and dynamic programming: a measure theoretic approach.* 1985.

15 F.J. van der Linden. *Euclidean rings with two infinite primes.* 1985.

16 R.J.P. Groothuizen. *Mixed elliptic-hyperbolic partial differential operators: a case-study in Fourier integral operators.* 1985.

17 H.M.M. ten Eikelder. *Symmetries for dynamical and Hamiltonian systems.* 1985.

18 A.D.M. Kester. *Some large deviation results in statistics.* 1985.

19 T.M.V. Janssen. *Foundations and applications of Montague grammar, part 1: Philosophy, framework, computer science.* 1986.

20 B.F. Schriever. *Order dependence.* 1986.

21 D.P. van der Vecht. *Inequalities for stopped Brownian motion.* 1986.

22 J.C.S.P. van der Woude. *Topological dynamix.* 1986.

23 A.F. Monna. *Methods, concepts and ideas in mathematics: aspects of an evolution.* 1986.

24 J.C.M. Baeten. *Filters and ultrafilters over definable subsets of admissible ordinals.* 1986.

25 A.W.J. Kolen. *Tree network and planar rectilinear location theory.* 1986.

26 A.H. Veen. *The misconstrued semicolon: Reconciling imperative languages and dataflow machines.* 1986.

27 A.J.M. van Engelen. *Homogeneous zero-dimensional absolute Borel sets.* 1986.

28 T.M.V. Janssen. *Foundations and applications of Montague grammar, part 2: Applications to natural language.* 1986.

29 H.L. Trentelman. *Almost invariant subspaces and high gain feedback.* 1986.

30 A.G. de Kok. *Production-inventory control models: approximations and algorithms.* 1987.

31 E.E.M. van Berkum. *Optimal paired comparison designs for factorial experiments.* 1987.

32 J.H.J. Einmahl. *Multivariate empirical processes.* 1987.

33 O.J. Vrieze. *Stochastic games with finite state and action spaces.* 1987.

34 P.H.M. Kersten. *Infinitesimal symmetries: a computational approach.* 1987.

35 M.L. Eaton. *Lectures on topics in probability inequalities.* 1987.

36 A.H.P. van der Burgh, R.M.M. Mattheij (eds.). *Proceedings of the first international conference on industrial and applied mathematics (ICIAM 87).* 1987.

37 L. Stougie. *Design and analysis of algorithms for stochastic integer programming.* 1987.

38 J.B.G. Frenk. *On Banach algebras, renewal measures and regenerative processes.* 1987.

39 H.J.M. Peters, O.J. Vrieze (eds.). *Surveys in game theory and related topics.* 1987.

40 J.L. Geluk, L. de Haan. *Regular variation, extensions and Tauberian theorems.* 1987.

41 Sape J. Mullender (ed.). *The Amoeba distributed operating system: Selected papers 1984-1987.* 1987.

42 P.R.J. Asveld, A. Nijholt (eds.). *Essays on concepts, formalisms, and tools.* 1987.

43 H.L. Bodlaender. *Distributed computing: structure and complexity.* 1987.

44 A.W. van der Vaart. *Statistical estimation in large parameter spaces.* 1988.

45 S.A. van de Geer. *Regression analysis and empirical processes.* 1988.

46 S.P. Spekreijse. *Multigrid solution of the steady Euler equations.* 1988.

47 J.B. Dijkstra. *Analysis of means in some non-standard situations.* 1988.

48 F.C. Drost. *Asymptotics for generalized chi-square goodness-of-fit tests.* 1988.

49 F.W. Wubs. *Numerical solution of the shallow-water equations.* 1988.

50 F. de Kerf. *Asymptotic analysis of a class of perturbed Korteweg-de Vries initial value problems.* 1988.

51 P.J.M. van Laarhoven. *Theoretical and computational aspects of simulated annealing.* 1988.

52 P.M. van Loon. *Continuous decoupling transformations for linear boundary value problems.* 1988.

53 K.C.P. Machielsen. *Numerical solution of optimal control problems with state constraints by sequential quadratic programming in function space.* 1988.

54 L.C.R.J. Willenborg. *Computational aspects of survey data processing.* 1988.

55 G.J. van der Steen. *A program generator for recognition, parsing and transduction with syntactic patterns.* 1988.

56 J.C. Ebergen. *Translating programs into delay-insensitive circuits.* 1989.

57 S.M. Verduyn Lunel. *Exponential type calculus for linear delay equations.* 1989.

58 M.C.M. de Gunst. *A random model for plant cell population growth.* 1989.

59 D. van Dulst. *Characterizations of Banach spaces not containing l^1.* 1989.

60 H.E. de Swart. *Vacillation and predictability properties of low-order atmospheric spectral models.* 1989.

61 P. de Jong. *Central limit theorems for generalized multilinear forms.* 1989.

62 V.J. de Jong. *A specification system for statistical software.* 1989.

63 B. Hanzon. *Identifiability, recursive identification and spaces of linear dynamical systems, part I.* 1989.

64 B. Hanzon. *Identifiability, recursive identification and spaces of linear dynamical systems, part II.* 1989.

MATHEMATICAL CENTRE TRACTS

1 T. van der Walt. *Fixed and almost fixed points.* 1963.

2 A.R. Bloemena. *Sampling from a graph.* 1964.

3 G. de Leve. *Generalized Markovian decision processes, part I: model and method.* 1964.

4 G. de Leve. *Generalized Markovian decision processes, part II: probabilistic background.* 1964.

5 G. de Leve, H.C. Tijms, P.J. Weeda. *Generalized Markovian decision processes, applications.* 1970.

6 M.A. Maurice. *Compact ordered spaces.* 1964.

7 W.R. van Zwet. *Convex transformations of random variables.* 1964.

8 J.A. Zonneveld. *Automatic numerical integration.* 1964.

9 P.C. Baayen. *Universal morphisms.* 1964.

10 E.M. de Jager. *Applications of distributions in mathematical physics.* 1964.

11 A.B. Paalman-de Miranda. *Topological semigroups.* 1964.

12 J.A.Th.M. van Berckel, H. Brandt Corstius, R.J. Mokken, A. van Wijngaarden. *Formal properties of newspaper Dutch.* 1965.

13 H.A. Lauwerier. *Asymptotic expansions.* 1966, out of print; replaced by MCT 54.

14 H.A. Lauwerier. *Calculus of variations in mathematical physics.* 1966.

15 R. Doornbos. *Slippage tests.* 1966.

16 J.W. de Bakker. *Formal definition of programming languages with an application to the definition of ALGOL 60.* 1967.

17 R.P. van de Riet. *Formula manipulation in ALGOL 60, part 1.* 1968.

18 R.P. van de Riet. *Formula manipulation in ALGOL 60, part 2.* 1968.

19 J. van der Slot. *Some properties related to compactness.* 1968.

20 P.J. van der Houwen. *Finite difference methods for solving partial differential equations.* 1968.

21 E. Wattel. *The compactness operator in set theory and topology.* 1968.

22 T.J. Dekker. *ALGOL 60 procedures in numerical algebra, part 1.* 1968.

23 T.J. Dekker, W. Hoffmann. *ALGOL 60 procedures in numerical algebra, part 2.* 1968.

24 J.W. de Bakker. *Recursive procedures.* 1971.

25 E.R. Paërl. *Representations of the Lorentz group and projective geometry.* 1969.

26 European Meeting 1968. *Selected statistical papers, part I.* 1968.

27 European Meeting 1968. *Selected statistical papers, part II.* 1968.

28 J. Oosterhoff. *Combination of one-sided statistical tests.* 1969.

29 J. Verhoeff. *Error detecting decimal codes.* 1969.

30 H. Brandt Corstius. *Exercises in computational linguistics.* 1970.

31 W. Molenaar. *Approximations to the Poisson, binomial and hypergeometric distribution functions.* 1970.

32 L. de Haan. *On regular variation and its application to the weak convergence of sample extremes.* 1970.

33 F.W. Steutel. *Preservation of infinite divisibility under mixing and related topics.* 1970.

34 I. Juhász, A. Verbeek, N.S. Kroonenberg. *Cardinal functions in topology.* 1971.

35 M.H. van Emden. *An analysis of complexity.* 1971.

36 J. Grasman. *On the birth of boundary layers.* 1971.

37 J.W. de Bakker, G.A. Blaauw, A.J.W. Duijvestijn, E.W. Dijkstra, P.J. van der Houwen, G.A.M. Kamsteeg-Kemper, F.E.J. Kruseman Aretz, W.L. van der Poel, J.P. Schaap-Kruseman, M.V. Wilkes, G. Zoutendijk. *MC-25 Informatica Symposium.* 1971.

38 W.A. Verloren van Themaat. *Automatic analysis of Dutch compound words.* 1972.

39 H. Bavinck. *Jacobi series and approximation.* 1972.

40 H.C. Tijms. *Analysis of (s,S) inventory models.* 1972.

41 A. Verbeek. *Superextensions of topological spaces.* 1972.

42 W. Vervaat. *Success epochs in Bernoulli trials (with applications in number theory).* 1972.

43 F.H. Ruymgaart. *Asymptotic theory of rank tests for independence.* 1973.

44 H. Bart. *Meromorphic operator valued functions.* 1973.

45 A.A. Balkema. *Monotone transformations and limit laws.* 1973.

46 R.P. van de Riet. *ABC ALGOL, a portable language for formula manipulation systems, part 1: the language.* 1973.

47 R.P. van de Riet. *ABC ALGOL, a portable language for formula manipulation systems, part 2: the compiler.* 1973.

48 F.E.J. Kruseman Aretz, P.J.W. ten Hagen, H.L. Oudshoorn. *An ALGOL 60 compiler in ALGOL 60, text of the MC-compiler for the EL-X8.* 1973.

49 H. Kok. *Connected orderable spaces.* 1974.

50 A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, R.G. Fisker (eds.). *Revised report on the algorithmic language ALGOL 68.* 1976.

51 A. Hordijk. *Dynamic programming and Markov potential theory.* 1974.

52 P.C. Baayen (ed.). *Topological structures.* 1974.

53 M.J. Faber. *Metrizability in generalized ordered spaces.* 1974.

54 H.A. Lauwerier. *Asymptotic analysis, part 1.* 1974.

55 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 1: theory of designs, finite geometry and coding theory.* 1974.

56 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 2: graph theory, foundations, partitions and combinatorial geometry.* 1974.

57 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 3: combinatorial group theory.* 1974.

58 W. Albers. *Asymptotic expansions and the deficiency concept in statistics.* 1975.

59 J.L. Mijnheer. *Sample path properties of stable processes.* 1975.

60 F. Göbel. *Queueing models involving buffers.* 1975.

63 J.W. de Bakker (ed.). *Foundations of computer science.* 1975.

64 W.J. de Schipper. *Symmetric closed categories.* 1975.

65 J. de Vries. *Topological transformation groups, 1: a categorical approach.* 1975.

66 H.G.J. Pijls. *Logically convex algebras in spectral theory and eigenfunction expansions.* 1976.

68 P.P.N. de Groen. *Singularly perturbed differential operators of second order.* 1976.

69 J.K. Lenstra. *Sequencing by enumerative methods.* 1977.

70 W.P. de Roever, Jr. *Recursive program schemes: semantics and proof theory.* 1976.

71 J.A.E.E. van Nunen. *Contracting Markov decision processes.* 1976.

72 J.K.M. Jansen. *Simple periodic and non-periodic Lamé functions and their applications in the theory of conical waveguides.* 1977.

73 D.M.R. Leivant. *Absoluteness of intuitionistic logic.* 1979.

74 H.J.J. te Riele. *A theoretical and computational study of generalized aliquot sequences.* 1976.

75 A.E. Brouwer. *Treelike spaces and related connected topological spaces.* 1977.

76 M. Rem. *Associons and the closure statement.* 1976.

77 W.C.M. Kallenberg. *Asymptotic optimality of likelihood ratio tests in exponential families.* 1978.

78 E. de Jonge, A.C.M. van Rooij. *Introduction to Riesz spaces.* 1977.

79 M.C.A. van Zuijlen. *Emperical distributions and rank statistics.* 1977.

80 P.W. Hemker. *A numerical study of stiff two-point boundary problems.* 1977.

81 K.R. Apt, J.W. de Bakker (eds.). *Foundations of computer science II, part 1.* 1976.

82 K.R. Apt, J.W. de Bakker (eds.). *Foundations of computer science II, part 2.* 1976.

83 L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH system.* 1979.

84 H.L.L. Busard. *The translation of the elements of Euclid from the Arabic into Latin by Hermann of Carinthia (?), books vii-xii.* 1977.

85 J. van Mill. *Supercompactness and Wallman spaces.* 1977.

86 S.G. van der Meulen, M. Veldhorst. *Torrix I, a programming system for operations on vectors and matrices over arbitrary fields and of variable size.* 1978.

88 A. Schrijver. *Matroids and linking systems.* 1977.

89 J.W. de Roever. *Complex Fourier transformation and analytic functionals with unbounded carriers.* 1978.

90 L.P.J. Groenewegen. *Characterization of optimal strategies in dynamic games.* 1981.

91 J.M. Geysel. *Transcendence in fields of positive characteristic.* 1979.

92 P.J. Weeda. *Finite generalized Markov programming.* 1979.

93 H.C. Tijms, J. Wessels (eds.). *Markov decision theory.* 1977.

94 A. Bijlsma. *Simultaneous approximations in transcendental number theory.* 1978.

95 K.M. van Hee. *Bayesian control of Markov chains.* 1978.

96 P.M.B. Vitányi. *Lindenmayer systems: structure, languages, and growth functions.* 1980.

97 A. Federgruen. *Markovian control problems; functional equations and algorithms.* 1984.

98 R. Geel. *Singular perturbations of hyperbolic type.* 1978.

99 J.K. Lenstra, A.H.G. Rinnooy Kan, P. van Emde Boas (eds.). *Interfaces between computer science and operations research.* 1978.

100 P.C. Baayen, D. van Dulst, J. Oosterhoff (eds.). *Proceedings bicentennial congress of the Wiskundig Genootschap, part 1.* 1979.

101 P.C. Baayen, D. van Dulst, J. Oosterhoff (eds.). *Proceedings bicentennial congress of the Wiskundig Genootschap, part 2.* 1979.

102 D. van Dulst. *Reflexive and superreflexive Banach spaces.* 1978.

103 K. van Harn. *Classifying infinitely divisible distributions by functional equations.* 1978.

104 J.M. van Wouwe. *Go-spaces and generalizations of metrizability.* 1979.

105 R. Helmers. *Edgeworth expansions for linear combinations of order statistics.* 1982.

106 A. Schrijver (ed.). *Packing and covering in combinatorics.* 1979.

107 C. den Heijer. *The numerical solution of nonlinear operator equations by imbedding methods.* 1979.

108 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science III, part 1.* 1979.

109 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science III, part 2.* 1979.

110 J.C. van Vliet. *ALGOL 68 transput, part I: historical review and discussion of the implementation model.* 1979.

111 J.C. van Vliet. *ALGOL 68 transput, part II: an implementation model.* 1979.

112 H.C.P. Berbee. *Random walks with stationary increments and renewal theory.* 1979.

113 T.A.B. Snijders. *Asymptotic optimality theory for testing problems with restricted alternatives.* 1979.

114 A.J.E.M. Janssen. *Application of the Wigner distribution to harmonic analysis of generalized stochastic processes.* 1979.

115 P.C. Baayen, J. van Mill (eds.). *Topological structures II, part 1.* 1979.

116 P.C. Baayen, J. van Mill (eds.). *Topological structures II, part 2.* 1979.

117 P.J.M. Kallenberg. *Branching processes with continuous state space.* 1979.

118 P. Groeneboom. *Large deviations and asymptotic efficiencies.* 1980.

119 F.J. Peters. *Sparse matrices and substructures, with a novel implementation of finite element algorithms.* 1980.

120 W.P.M. de Ruyter. *On the asymptotic analysis of large-scale ocean circulation.* 1980.

121 W.H. Haemers. *Eigenvalue techniques in design and graph theory.* 1980.

122 J.C.P. Bus. *Numerical solution of systems of nonlinear equations.* 1980.

123 I. Yuhász. *Cardinal functions in topology - ten years later.* 1980.

124 R.D. Gill. *Censoring and stochastic integrals.* 1980.

125 R. Eising. *2-D systems, an algebraic approach.* 1980.

126 G. van der Hoek. *Reduction methods in nonlinear programming.* 1980.

127 J.W. Klop. *Combinatory reduction systems.* 1980.

128 A.J.J. Talman. *Variable dimension fixed point algorithms and triangulations.* 1980.

129 G. van der Laan. *Simplicial fixed point algorithms.* 1980.

130 P.J.W. ten Hagen, T. Hagen, P. Klint, H. Noot, H.J. Sint, A.H. Veen. *ILP: intermediate language for pictures.* 1980.

131 R.J.R. Back. *Correctness preserving program refinements: proof theory and applications.* 1980.

132 H.M. Mulder. *The interval function of a graph.* 1980.

133 C.A.J. Klaassen. *Statistical performance of location estimators.* 1981.

134 J.C. van Vliet, H. Wupper (eds.). *Proceedings international conference on ALGOL 68.* 1981.

135 J.A.G. Groenendijk, T.M.V. Janssen, M.J.B. Stokhof (eds.). *Formal methods in the study of language, part I.* 1981.

136 J.A.G. Groenendijk, T.M.V. Janssen, M.J.B. Stokhof (eds.). *Formal methods in the study of language, part II.* 1981.

137 J. Telgen. *Redundancy and linear programs.* 1981.

138 H.A. Lauwerier. *Mathematical models of epidemics.* 1981.

139 J. van der Wal. *Stochastic dynamic programming, successive approximations and nearly optimal strategies for Markov decision processes and Markov games.* 1981.

140 J.H. van Geldrop. *A mathematical theory of pure exchange economies without the no-critical-point hypothesis.* 1981.

141 G.E. Welters. *Abel-Jacobi isogenies for certain types of Fano threefolds.* 1981.

142 H.R. Bennett, D.J. Lutzer (eds.). *Topology and order structures, part 1.* 1981.

143 J.M. Schumacher. *Dynamic feedback in finite- and infinite-dimensional linear systems.* 1981.

144 P. Eijgenraam. *The solution of initial value problems using interval arithmetic; formulation and analysis of an algorithm.* 1981.

145 A.J. Brentjes. *Multi-dimensional continued fraction algorithms.* 1981.

146 C.V.M. van der Mee. *Semigroup and factorization methods in transport theory.* 1981.

147 H.H. Tigelaar. *Identification and informative sample size.* 1982.

148 L.C.M. Kallenberg. *Linear programming and finite Markovian control problems.* 1983.

149 C.B. Huijsmans, M.A. Kaashoek, W.A.J. Luxemburg, W.K. Vietsch (eds.). *From A to Z, proceedings of a symposium in honour of A.C. Zaanen.* 1982.

150 M. Veldhorst. *An analysis of sparse matrix storage schemes.* 1982.

151 R.J.M.M. Does. *Higher order asymptotics for simple linear rank statistics.* 1982.

152 G.F. van der Hoeven. *Projections of lawless sequences.* 1982.

153 J.P.C. Blanc. *Application of the theory of boundary value problems in the analysis of a queueing model with paired services.* 1982.

154 H.W. Lenstra, Jr., R. Tijdeman (eds.). *Computational methods in number theory, part I.* 1982.

155 H.W. Lenstra, Jr., R. Tijdeman (eds.). *Computational methods in number theory, part II.* 1982.

156 P.M.G. Apers. *Query processing and data allocation in distributed database systems.* 1983.

157 H.A.W.M. Kneppers. *The covariant classification of two-dimensional smooth commutative formal groups over an algebraically closed field of positive characteristic.* 1983.

158 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science IV, distributed systems, part 1.* 1983.

159 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science IV, distributed systems, part 2.* 1983.

160 A. Rezus. *Abstract AUTOMATH.* 1983.

161 G.F. Helminck. *Eisenstein series on the metaplectic group, an algebraic approach.* 1983.

162 J.J. Dik. *Tests for preference.* 1983.

163 H. Schippers. *Multiple grid methods for equations of the second kind with applications in fluid mechanics.* 1983.

164 F.A. van der Duyn Schouten. *Markov decision processes with continuous time parameter.* 1983.

165 P.C.T. van der Hoeven. *On point processes.* 1983.

166 H.B.M. Jonkers. *Abstraction, specification and implementation techniques, with an application to garbage collection.* 1983.

167 W.H.M. Zijm. *Nonnegative matrices in dynamic programming.* 1983.

168 J.H. Evertse. *Upper bounds for the numbers of solutions of diophantine equations.* 1983.

169 H.R. Bennett, D.J. Lutzer (eds.). *Topology and order structures, part 2.* 1983.