Distributed Operating Systems

ANDREW S. TANENBAUM and 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 are examined in some detail, namely, the Cambridge Distributed Computing System, Amoeba, V, and Eden.

Categories and Subject Descriptors: C.2.4 [**Computer-Communications Networks**]: Distributed Systems—*network operating system*; D.4.3 [**Operating Systems**]: File Systems Management—*distributed file systems*; D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*; D.4.6 [**Operating Systems**]: Security and Protection—*access controls*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

General Terms: Algorithms, Design, Experimentation, Reliability, Security

Additional Key Words and Phrases: File server

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 present a viewpoint widely held within academia about what is and is not a distributed system, we discuss numerous interesting design issues concerning them, and finally we 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 database 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 centralized operating system but runs on multiple, independent central processing units (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

[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.

CONTENTS

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, which we 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, instead of running part of a global, systemwide operating system.
- Each user normally works on his or her own machine; using a different machine invariably requires some kind of "remote login," instead of 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, instead of having file placement managed by the operating system.
- The system has little or no fault tolerance; if 1 percent of the personal computers crash, 1 percent of the users are out of business, instead of everyone simply being able to continue normal work, albeit with 1 percent worse performance.

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 with 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 by those who have.

Incremental growth is another plus; if you need 10 percent more computing power, you just add 10 percent 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 percent 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. There has been built more than one system requiring 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.

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

System Models

Various models have been suggested for building a distributed system. Most of them fall into one of three broad categories, which we 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., VAXs), 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 their location.

The processor pool model is the next evolutionary step after the workstation model. In a time-sharing 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 to await 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.

1. 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; Luderer et al. 1981; Wambecq 1983]. Wupit [1983] gives a good survey of these systems, which we shall 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.

1.1 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);
open(''machine1!pathname'', READ);
      or
open(''/../machine1/pathname'', READ);
```

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 upward 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 "machine 1" may really be any directory, so one can attach a machine as a leaf anywhere in the hierarchy, not just at the top.

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 they are 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 among machines to keep all the disks equally full and busy, and that the system can maintain

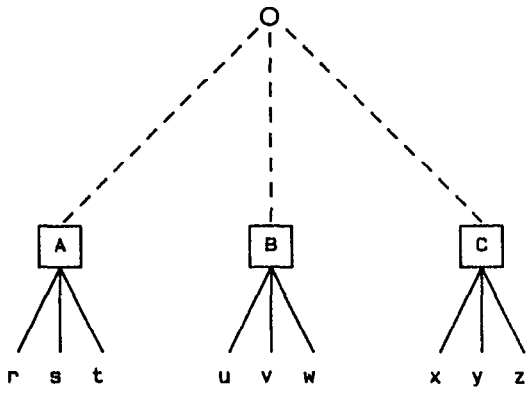

Figure 1. A (virtual) superdirectory above the root directory provides access to remote files.

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.

1.2 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), for example, 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 to 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 that when a user with UID 12 on his or her home machine accesses a remote machine on which his or her 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 or her 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, systemwide mapping that is valid everywhere.

1.3 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 thus 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 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 a 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 that files that were in the parent process's 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 among 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 would intercept all the system calls and decide 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.

1.4 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 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 bit-mapped 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 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 M1:/usr/jim/file.c M2:/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 an 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 upward through the VFS layer to the top layer, where they are reinjected 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 in the same way as local requests.

2. 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 look at five issues that distributed systems' designers are faced with:

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

Although 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.

2.1 Communication Primitives

The computers forming a distributed system normally do not share primary memory, and so communication via shared memory techniques such as semaphores and monitors is 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 [Zimmermann 1980]. The seven layers are the physical layer, data-link layer, network layer, transport layer, session layer, presentation layer, and application layer. By 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 kilobytes per second), 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-megabyte-per second or faster local network, the price of the ISO model is generally too high. Nearly all the experimental distributed systems discussed in the literature thus far have opted for a different, much simpler model, so we do not mention the ISO model further in this paper.

2.1.1 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 tear down is required.

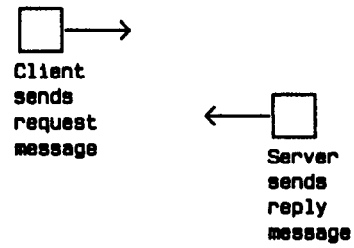

Figure 2. Client–server model of communication.

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 versus reliable and nonblocking versus blocking 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 acknowledgments internally, so that when SEND terminates, the program is sure that the message has been received and acknowledged.

Blocking versus 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 copy is made, any changes the program makes to the data before or (heaven forbid) *while* they are 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 arrives. The advantage of these nonblocking primitives is that they provide the maximum flexibility: Programs can

compute and perform message I/O in parallel in any way they want.

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 acknowledgment 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 a 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 versus 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 using reliable, blocking primitives, one can create some order in the chaos.

2.1.2 Remote Procedure Call (RPC)

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 call*" and has been widely discussed [Birrell and Nelson 1984; Nelson 1981; Spector 1982]. The idea is to make the semantics of inter-machine communication as similar as possible to normal procedure calls because the latter is familiar and 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).

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, and 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.

Remote Procedure Call 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 they go. Passing reference parameters (pointers) over the network is not so easy. One needs a unique, systemwide 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.

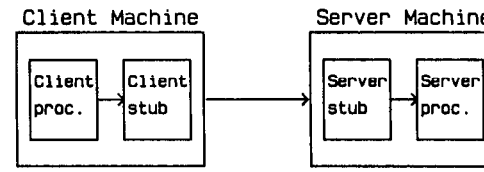

Figure 3. Remote procedure call.

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 the 1's complement 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 lets 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 it is done, with remote procedure call (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 that the receiver at least 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 sub-

sequent writes to that file 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 disadvantages, RPC remains an interesting form of communication, and much current research is being addressed toward improving it and solving the various problems discussed above.

2.1.3 Error Handling

Error handling in distributed systems is radically different from that 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 time-out is built in. However, such a time-out 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 $A \mid B$ 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 acknowledgment.)

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. This is a case in which it works: Client asks to read block 7 of some file. This is a case in which 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 that remote procedure calls are executed exactly once, and they 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 whether 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 cause events external to the computers.

2.1.4 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 1984a]. For example, a remote procedure call system that first copies each message from the user to the stub, from the stub to the kernel, and finally from the kernel to the network interface board requires three copies on the sending side, and probably three more on the receiving side, for a total of six. If the call is to a remote file server to write a 1K block of data to disk, at a copy time of 1 microsecond per byte, 6 milliseconds are needed just for copying, which puts an upper limit of 167 calls per second, or a throughput of 167 kilobytes per second. When other sources of overhead are considered (e.g., the reply message, the time waiting for access to the network, transmission time), achieving even 80 kilobytes per second 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 ten maximum-length packets, degrading response time; so the optimum size depends on the work load.

Virtual Circuits versus 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 acknowledgments at the highest level of protocol, the lower levels need not be 100 percent 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 post, telephone, and telegraph (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 opts for building RPC 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 Figure 4a. 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 4b. However, a problem arises when the REPLY can be delayed for a long time. For example, when a login process makes an RPC 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 whether the message has 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 acknowledgment of the previous REPLY (see Figure 4c). 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"

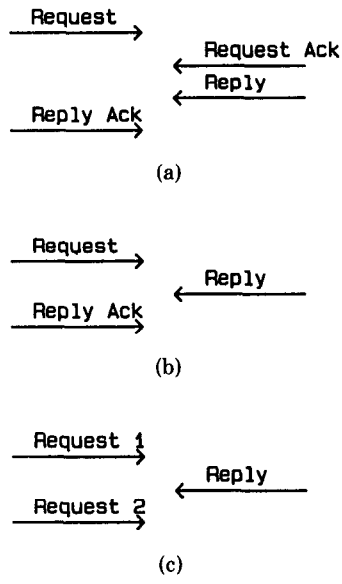

Figure 4. Remote procedure call (a) with individual acknowledgments per message, (b) with the reply as the request acknowledgment, (c) with no explicit acknowledgments.

the 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 that 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 reenable the receiving hardware. Until the receiver is reenabled, 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 (unbeknown to *B*) deaf file server. Even if the server reenables halfway through *B*'s message, the message will be rejected owing 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.

2.2 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.

2.2.1 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 that process is running. The mapping need not always be unique, for example, if there are multiple processes prepared to offer the same service.

2.2.2 Name Servers

In centralized systems, the problem of naming can be effectively handled in a straightforward way. The system maintains a table or database 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 database 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 owing 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

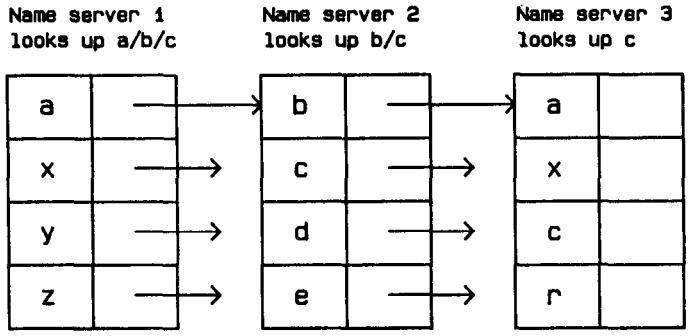

Figure 5. Distributing the lookup of *a/b/c* over three name servers.

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 three 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.

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 that it is supposed to be offering so that it can figure out what program to reboot.

2.3 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 whether a slot is free. A distributed operating system, on the other hand, has a much harder job of finding out whether 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 look at some work that has been done, including distributed process management and scheduling.

2.3.1 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 a replacement.

Although this scheme is not completely distributed, it is feasible and works well in practice. 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 $R \geq S$, 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 upward 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 so 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.

2.3.2 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 that the whole group runs at once. In this section we 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

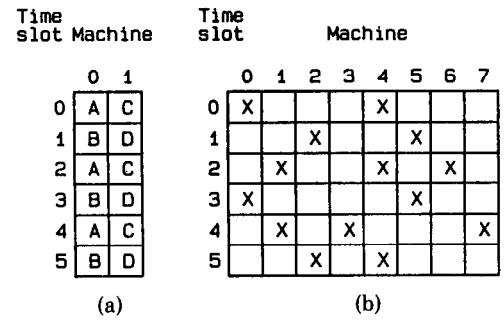

Figure 6. (a) Two jobs running out of phase with each other. (b) Scheduling matrix for eight machines, each with six time slots. The X's indicated allocated slots.

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-millisecond time slices, with A and C running in the even slices, and B and D running in the odd ones, as shown in Figure 6a. 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 milliseconds, 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 milliseconds before A gets the reply and can proceed. The net result is one message exchange every 200 milliseconds. What is needed is a way to ensure that processes that communicate frequently run simultaneously.

Although it is difficult to determine dynamically 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 are 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 6b. 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 6b, 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.

2.3.3 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 that closely related groups of processes can be placed on the same machine to reduce interprocess communication costs [Chow and Abraham 1982; Chu et al. 1980; Gylys and Edwards 1976; Lo 1984; Stone 1977, 1978; Stone and Bokhari 1978]. 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 trade-offs.

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

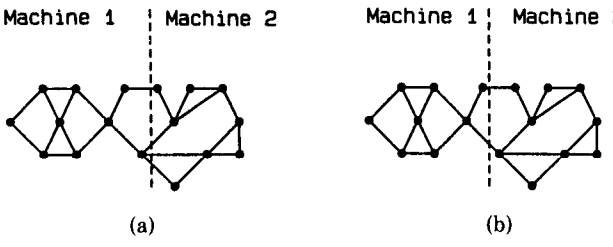

Figure 7. Two ways of statically allocating processes (nodes in the graph) to machines. Arcs show which pairs of processes communicate.

people making real systems who care less about optimality than about 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 7a depicts a graph of interacting processors with one possible partitioning of the processes between two machines. Figure 7b 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], Chow and Kohler [1979], Stone [1977, 1978], Stone and Bokhari [1978], and 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 estimate its own load continually, 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 that all its processes will need to finish. The residual time can be estimated mostly simply by 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 seconds than against 100 processes that each need 1 second.

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 [Barak and Shiloh 1985; Smith 1979] is for each processor periodically to 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, instead of 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.

2.3.4 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 all of some set of processes are 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 database 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 one 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.

2.4 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 one's data. Availability has to do with the system being up when it is needed. 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.

Distributed systems are potentially more reliable than a centralized system because 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.

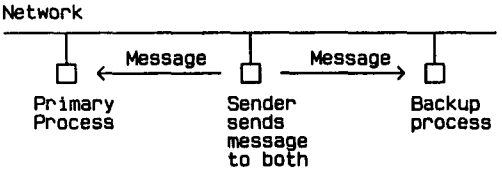

Figure 8. Each process has its own backup process.

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.

2.4.1 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.

Thus, if one process crashes because of 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.

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 that 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

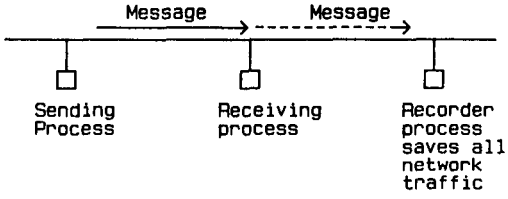

Figure 9. A recorder process copies and stores all network traffic without affecting the sender and receiver.

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 that a processor crashes because of a software bug. Both the schemes discussed above [Borg et al. 1983; Powell and Presotto 1983] 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 apply only to tolerance of hardware errors. It is also possible, however, 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 by Avizienis and Chen [1977], Avizienis and Kelly [1984], and Anderson and Lee [1981].

2.4.2 Atomic Transactions

When multiple users on several machines are concurrently updating a distributed database 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 database 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 databases 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 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 a 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 whether or not a block is valid, 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 is replaced by a copy of the first. 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, e.g., Fridrich and Older [1981, 1984], Mitchell and Dion [1982], Brown et al. [1985], Popek et al. [1981], and Reed and Svobodova [1981]).

2.5 Services

In a distributed system, it is natural for user-level server processes to provide functions that have been traditionally 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 look at some of these services, but first we look at how services and servers can be structured.

2.5.1 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 in which the server is blocked waiting for the remote disk to reply, it might have been able to service the next ten 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

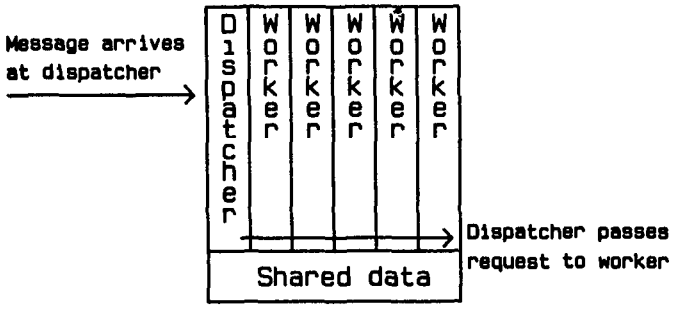

Figure 10. The dispatcher task waits for requests and passes them on to the worker tasks.

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 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 important, 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.

2.5.2 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 Tanenbaum [1985], Reed and Svobodova [1981], Satyanarayanan et al. [1985], Schroeder et

al. [1985], Sturgis et al. [1980], Svobodova [1981], and 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 that 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. The gain, however, is increased run-time 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, five 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 time-out 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 stateless? 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 stateless 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 stateless 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 stateless 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 stateless model each request is completely self-contained (file name, file position, etc.), so a newly 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 reorder messages, that is, deliver the second message before the first. 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 three categories: owner, group, and others. With capabilities, a user must present a special "ticket" on each file access proving that he or she 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 or she 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, 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 by Birrell [1985].

With capabilities, the protection normally results from 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 or her

machine. A possible solution is to encrypt the capabilities, as discussed by Mullender and Tanenbaum [1984, 1985, 1986] and 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 an 8-megahertz 68000 processor and a 10-megabyte-per-second Ethernet, they observe that reading a 512-byte block from the local machine takes 1.3 milliseconds and from a remote machine 5.7 milliseconds, 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 milliseconds versus 60 milliseconds 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 by 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 because of 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 by Brown et al. [1985], Dion [1980], Mitchell and Dion [1982], Needham and Herbert [1982], Reed and Svobodova [1981], Sturgis et al. [1980], and 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 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 that 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 by 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.

2.5.3 Print Service

Compared with 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, 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 acknowledgment 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 by Janson et al. [1983] and Needham and Herbert [1982].

2.5.4 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 very large-scale integration (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.

2.5.5 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 hard wired to the computer that 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.

2.5.6 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. When a site has dozens of computers spread over multiple local networks, however, 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 systemwide. Almes et al. [1985] discuss how mail is handled in the Eden system.

2.5.7 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 queuing 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, e.g., Jefferson [1985], Lamport [1978, 1984], Marzullo and Owicki [1985], Reed [1983], and Reif and Spirakis [1984]).

2.5.8 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 read-only memory (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 others are still working properly.

2.5.9 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.

3. 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, 1984; Liskov and Scheifler 1982; Oki et al. 1985], Chorus [Zimmermann, 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 1978, 1979], and the work at Xerox Palo Alto Research Center [Birrell 1985; Birrell and Nelson 1984; Birrell et al. 1984; Boggs et al. 1980; Brown et al. 1985; Swinehart et al. 1979].

The systems we 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.

3.1 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 acknowledgments), several block-oriented protocols have been developed for reliably exchanging 2K packets by accumulating 1024 minipackets. The nominal ring bandwidth is 10 megabytes per second, but since each minipacket has 2 bytes of data and 3 bytes of overhead, the effective bandwidth is 4 megabytes per second.

The Cambridge ring project was very successful, with copies of the ring currently in operation at many universities and companies in the United Kingdom 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 or she 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.

3.1.1 Communication Primitives

Owing to the evolution from network to distributed system described earlier, the communication primitives are usually described as network protocols rather than 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 then having the server send a reply. Some machines are multiprogrammed, so that the second minipacket 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.

3.1.2 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 database is relatively static, from time to time names must be added or deleted to the name server's database. 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 that the service can check to see whether 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 or her session key (a big random number), the user's class (e.g., faculty, student), and a control key, as shown in Figure 11.

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.

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.

3.1.3 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 for allocation. Various defaults are available, so, for example, a user can specify wanting 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 that users have the same control over a processor bank machine as they would over real personal computers on their desks.

3.1.4 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

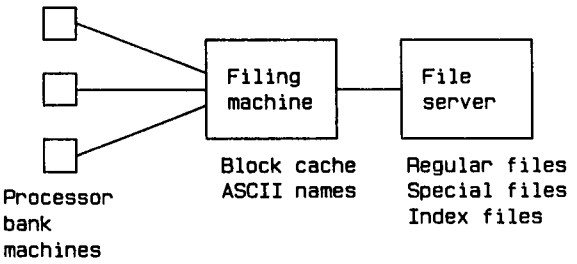

Figure 12. The filing machine is positioned between the users and the file server. It maintains a block cache and handles ASCII names.

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.

3.1.5 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 Cambridge Computing Laboratory that does not have 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 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 indexes 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 directo-

ries 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 lookup occurring in the filing machine now. Owing to the large cache, most of the relevant directories are likely to be already 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 because of lack of buffer space on the user side. The resulting improvements were substantial.

3.1.6 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 be actually 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-megabit-per-second) lines.

3.2 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-megabytes-per-second local network.

The Amoeba architecture consists of four principal components, as shown in Figure 13. First are the workstations, one per user,

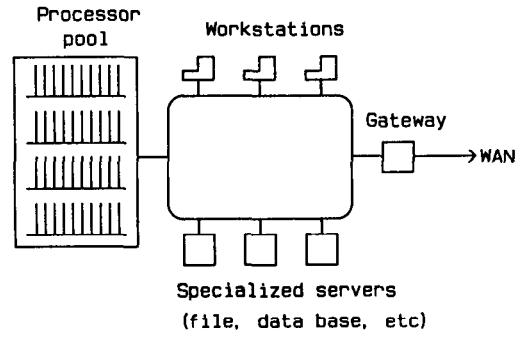

Figure 13. The Amoeba architecture.

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, database 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 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.

3.2.1 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 32 kilobytes to a server and then block waiting for the result. Servers use GET_REQUEST and PUT_REPLY 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, acknowledgment 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 database 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 or she 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 that 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."

3.2.2 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, and devices, 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, or starting or stopping a process. The number and types 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 in which 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

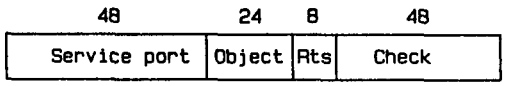

Figure 14. An Amoeba capability.

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 (comparable to 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.

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 that 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 whether the requested operation is permitted. More details about protection of capabilities can be found in Mullender and Tanenbaum [1984, 1986] and 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, and 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.

3.2.3 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's 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 important, 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 ten 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 processors are free and which 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 batchlike 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 or she 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, and 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.

3.2.4 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 percent 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 setup; 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 that 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.

3.2.5 Services

Amoeba has several kinds of block, file, and directory services. 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 relink 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 database 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 greater detail by Mullender and Tanenbaum [1985].

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

3.2.6 Implementation

The Amoeba kernel has been ported to five different CPUs: 68010, NS32016, 8088, VAX, and PDP-11. 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 milliseconds (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 per second, which is about 20 percent 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.

3.3 The V Kernel

The V kernel is a research project on distributed systems at Stanford University under the direction of David Cheriton [Cheriton 1984a; Cheriton and Mann 1984; Cheriton and Zwaenepoel 1984a, 1984b]. 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 back plane, analogous to the Multibus or S-100 bus back planes. The function of a back plane 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 or her own. This view is in contrast to the "centrally planned economy" model of most

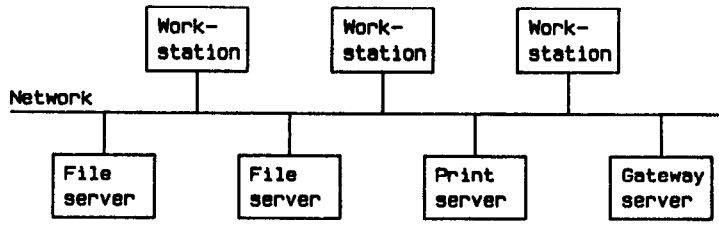

Figure 15. A typical V configuration.

time-sharing 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.

3.3.1 Communication Primitives

The V communication primitives have been designed in accordance with the back-plane 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 pseudopointer to one of the client's memory segments. This pseudopointer 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 pseudopointers 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 has been piggybacked onto the message, it will have a place to go. The REPLY call can also provide a segment buffer for the case in which the client provides a pseudopointer that the server can use to return results exceeding 32 bytes.

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

3.3.2 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 (Server-Name, 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.

3.3.3 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 proved general enough to handle disks, printers, terminals, and even a mouse.

3.3.4 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. V, however, 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 afterward.

3.3.5 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 milliseconds 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 milliseconds. Given that the access time of the small Winchester disks used on personal computers is rarely better than 40 milliseconds, it is clear that the V implementation of diskless workstations with a fast (18-millisecond) 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.

3.3.6 Implementation

The V kernel has been up and running at Stanford University since September 1982. It runs on SUN Microsystems 68000-based workstations, connected by 3-megabit-per-second and 10-megabit-per-second 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.

3.4 The Eden Project

The goal of the Eden system [Almes et al. 1985; Black 1983, 1985; 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 and 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 redesigning. This would have been much more difficult in a bottom-up, hardware-driven approach.

3.4.1 Communication 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 others 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.

3.4.2 Naming and Protection

Naming and protection in Eden are 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.

3.4.3 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 run-time 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 systematically to find all capabilities by scanning the kernel's data structures.

3.4.4 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 that 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 acknowledgment back to the sender, to ensure that letters are never lost because of 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).

3.4.5 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 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 with the traditional one-file-server-for-all-files way of doing things. There are two main advantages. First, the complicated, multithreaded 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.

3.4.6 *Implementation*

Eden has had a somewhat tortuous implementation history. The initial version was designed to be written in Ada[4] 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 to 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 VAX). 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 that would provide automatic stub generation, better type checking, and a more convenient way of dealing with concurrency.

This reevaluation 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

[4] Ada is a trademark of the U.S. Department of Defense.

amount of effort spent trying to convince UNIX to do things against its will.

3.5 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 summarize and compare the four systems with respect to the main design issues that we have been discussing.

3.5.1 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 32 kilobytes (with the kernel-handling message fragmentation and reassembly), as well as acknowledgments 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 first 1K is piggybacked onto the REQUEST itself.

Eden comes closest to a true RPC mechanism, including having a language and compiler 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.

3.5.2 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 that 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 that 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, and 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.

3.5.3 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 login 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. The user may load any operating system that he or she chooses in this machine.

Amoeba also has a pool of processors, but these are allocated dynamically. A user running "make" might be allocated ten processors to compile ten files; afterward, all the processors would go back into the pool. Amoeba also provides a way for processes to create segments on any machine (assuming that 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 systemwide resource management.

Eden has been built on top of existing operating systems, and therefore 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.

3.5.4 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 trade-off 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 has gone down, but it does automatically ensure that 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 and thus discouraging its frequent use.

3.5.5 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, which is provided by the filing machine or the user machines. The universal file server has regular and special files, of which the latter 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).

4. 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 that we draw is that distributed operating systems will be an interesting and fruitful area of research for a number of years to come.

ACKNOWLEDGMENTS

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., AND KIRK, P. 1982. Protocol architecture of the UNIVERSE project. In *Proceedings of the 6th International Conference on Computer Communication* (London, Sept. 7–10). International Conference for Computer Communication, pp. 379–383.

ALMES, G. T., BLACK, A. P., LAZOWSKA, E. D., AND NOE, J. D. 1985. The Eden system: A technical review. *IEEE Trans. Softw. Eng. SE-11* (Jan.), 43–59.

ANDERSON, T., AND LEE, P. A. 1981. *Fault Tolerance, Principles and Practice.* Prentice-Hall International, London.

AVIZIENIS, A., AND CHEN, L. 1977. On the implementation of N-version programming for software fault-tolerance during execution. In *Proceedings of the International Computer Software and Applications Conference.* IEEE, New York, pp. 149–155.

AVIZIENIS, A., AND KELLY, J. 1984. Fault tolerance by design diversity. *Computer 17* (Aug.), 66–80.

BAL, H. E., VAN RENESSE, R., AND TANENBAUM, A. S. 1985. A distributed, parallel, fault tolerant computing system. Rep. IR-106, Dept. of Mathematics and Computer Science, Vrije Univ., The Netherlands, Oct.

BALL, J. E., FELDMAN, J., LOW, R., RASHID, R., AND ROVNER, P. 1976. RIG, Rochester's intelligent gateway: System overview. *IEEE Trans. Softw. Eng. SE-2* (Dec.), 321–329.

BARAK, A., AND SHILOH, A. 1985. A distributed load-balancing policy for a multicomputer. *Softw. Pract. Exper. 15* (Sept.), 901–913.

BIRMAN, K. P., AND ROWE, L. A. 1982. A local network based on the UNIX operating system. *IEEE Trans. Softw. Eng. SE-8* (Mar.), 137–146.

BIRRELL, A. D. 1985. Secure communication using remote procedure calls. *ACM Trans. Comput. Syst. 3*, 1 (Feb.), 1–14.

BIRRELL, A. D., AND NEEDHAM, R. M. 1980. A universal file server. *IEEE Trans. Softw. Eng. SE-6*, (Sept.), 450–453.

BIRRELL, A. D., AND NELSON, B. J. 1984. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1 (Feb.), 39–59.

BIRRELL, A. D., LEVIN, R., NEEDHAM, R. M., AND SCHROEDER, M. 1982. Grapevine: An exercise in distributed computing. *Commun. ACM 25*, 4 (Apr.), 260–274.

BIRRELL, A. D., LEVIN, R., NEEDHAM, R. M., AND SCHROEDER, M. 1984. Experience with Grapevine: The growth of a distributed system. *ACM Trans. Comput. Syst. 2*, 1 (Feb.), 3–23.

BLACK, A. P. 1983. An asymmetric stream communications system. *Oper. Syst. Rev.* (ACM) *17*, 5, 4–10.

BLACK, A. P. 1985. Supporting distributed applications: Experience with Eden. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1–4). ACM, New York, pp. 181–193.

BOGGS, D. R., SCHOCH, J. F., TAFT, E. A., AND METCALFE, R. M. 1980. Pup: An internetwork architecture. *IEEE Trans. Commun. COM-28* (Apr.), 612–624.

BORG, A., BAUMBACH, J., AND GLAZER, S. 1983. A message system supporting fault tolerance. *Oper. Syst. Rev.* (ACM) *17*, 5, 90–99.

BROWN, M. R., KOLLING, K. N., AND TAFT. E. A. 1985. The Alpine file system. *ACM Trans. Comput. Syst. 3*, 4 (Nov.), 261–293.

BROWNBRIDGE, D. R., MARSHALL, L. F., AND RANDELL, B. 1982. The Newcastle connection—Or UNIXES of the world unite! *Softw. Pract. Exper. 12* (Dec.), 1147–1162.

BRYANT, R. M., AND FINKEL, R. A. 1981. A stable distributed scheduling algorithm. In *Proceedings of the 2nd International Conference on Distributed Computer Systems* (Apr.). IEEE, New York, pp. 314–323.

CHANDY, K. M., MISRA, J., AND HAAS, L. M. 1983. Distributed deadlock detection. *ACM Trans. Comput. Syst. 1*, 2 (May), 145–156.

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

CHERITON, D. R. 1984a. An experiment using registers for fast message-based interprocess communication. *Oper. Syst. Rev. 18* (Oct.), 12–20.

CHERITON, D. R. 1984b. The V kernel: A software base for distributed systems. *IEEE Softw. 1* (Apr.), 19–42.

CHERITON, D. R., AND MANN, T. P. 1984. Uniform access to distributed name interpretation in the

V system. In *Proceedings of the 4th International Conference on Distributed Computing Systems.* IEEE, New York, pp. 290–297.

CHERITON, D. R., AND ZWAENEPOEL, W. 1983. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the 9th Symposium on Operating System Principles.* ACM, New York, pp. 128–140.

CHERITON, D. R., AND ZWAENEPOEL, W. 1984. One-to-many interprocess communication in the V-system. In *SIGCOMM '84 Tutorials and Symposium on Communications Architectures and Protocols* (Montreal, Quebec, June 6–8). ACM, New York.

CHERITON, D. R., MALCOLM, M. A., MELEN, L. S., AND SAGER, G. R. 1979. Thoth, a portable real-time operating system. *Commun. ACM 22,* 2 (Feb.), 105–115.

CHESSON, G. 1975. The network UNIX system. In *Proceedings of the 5th Symposium on Operating Systems Principles* (Austin, Tex., Nov. 19–21). ACM, New York, pp. 60–66.

CHOW, T. C. K., AND ABRAHAM, J. A. 1982. Load balancing in distributed systems. *IEEE Trans. Softw. Eng. SE-8* (July), 401–412.

CHOW, Y. C., AND KOHLER, W. H. 1979. Models for dynamic load balancing in heterogeneous multiple processor systems. *IEEE Trans. Comput. C-28* (May), 354–361.

CHU, W. W., HOLLOWAY, L. J., MIN-TSUNG, L., AND EFE, K. 1980. Task allocation in distributed data processing. *Computer 13* (Nov.), 57–69.

CURTIS, R. S., AND WITTIE, L. D. 1984. Global naming in distributed systems. *IEEE Softw. 1,* 76–80.

DALAL, Y. K. 1977. Broadcast protocols in packet switched computer networks. Ph.D. Dissertation, Computer Science Dept., Stanford Univ., Stanford, Calif.

DELLAR, C. 1982. A file server for a network of low-cost personal microcomputers. *Softw. Pract. Exper. 12* (Nov.), 1051–1068.

DENNIS, J. B., AND VAN HORN, E. C. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM 9,* 3 (Mar.), 143–154.

DEWITT, D. J., FINKEL, R. A., AND SOLOMON, M. 1984. The CRYSTAL multicomputer: Design and implementation experience. Tech. Rep. TR-553, Computer Science Dept., Univ. of Wisconsin, Madison, Wis.

DION, J. 1980. The Cambridge file server. *Oper. Syst. Rev.* (ACM) *14* (Oct.), 41–49.

EFE, K. 1982. Heuristic models of task assignment scheduling in distributed systems. *Computer 15* (June), 50–56.

ESWARAN, K. P., GRAY, J. N., LORIE, J. N., AND TRAIGER, I. L. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM 19,* 11 (Nov.), 624–633.

FARBER, D. J., AND LARSON, K. C. 1972. The system architecture of the distributed computer system—The communications system. In *Proceedings of the Symposium on Computer Networks* (Brooklyn, Apr.). Polytechnic Inst. of Brooklyn, Brooklyn, N.Y.

FINKEL, R. A., SOLOMON, M. H., AND TISCHLER, R. 1979. The Roscoe resource manager. *COMPCON 79 Digest of Papers* (Feb.). IEEE, New York, pp. 88–91.

FITZGERALD, R., AND RASHID R. 1985. The integration of virtual memory management and interprocess communication in Accent. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1–4). ACM, New York, pp. 13–14.

FRIDRICH, M., AND OLDER, W. 1981. The Felix file server. In *Proceedings of the 8th Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 14–16). ACM, New York, pp. 37–44.

FRIDRICH, M., AND OLDER, W. 1984. HELIX: The architecture of a distributed file system. In *Proceedings of the 4th International Conference on Distributed Computing Systems.* IEEE, New York, pp. 422–431.

GAGLIANELLO, R. D., AND KATSEFF, H. P. 1985. Meglos: An operating system for a multiprocessor environment. In *Proceedings of the 5th International Conference on Distributed Computing Systems* (May). IEEE, New York, pp. 35–42.

GLIGOR, V. D., AND SHATTUCK, S. H. 1980. Deadlock detection in distributed systems. *IEEE Trans. Softw. Eng. SE-6* (Sept.), 435–440.

GYLYS, V. B., AND EDWARDS, J. A. 1976. Optimal partitioning of workload for distributed systems. In *Proceedings of COMPCON* (Sept.). IEEE, New York, pp. 353–357.

HWANG, K., CROFT, W. J., GOBLE, G. H. WAH, B. W., BRIGGS, F. A., SIMMONS, W. R., AND COATES, C. L. 1982. A UNIX-based local computer network. *Computer 15* (Apr.), 55–66.

ISLOOR, S. S., AND MARSLAND, T. A. 1978. An effective on-line deadlock detection technique for distributed database management systems. In *Proceedings of the International Computer and Software Application Conference.* IEEE, New York, pp. 283–288.

JANSON, P., SVOBODOVA, L., AND MAEHLE, E. 1983. Filing and printing services in a local area network. In *Proceedings of the 8th Data Communications Symposium* (Cape Cod, Mass., Oct. 3–6). IEEE, New York, pp. 211–219.

JEFFERSON, D. R. 1985. Virtual time. *ACM Trans. Program. Lang. Syst. 7* (July), 404–425.

JENSEN, E. D. 1978. The Honeywell experimental distributed processor—An overview of its objective, philosophy and architectural facilities. *Computer 11* (Jan), 28–38.

JESSOP, W. H., JACOBSON, D. M., NOE, J. D., BAER, J.-L., AND PU, C. 1982. The Eden transaction-based file system. In *Proceedings of the 2nd Sym-*

posium on Reliability in Distributed Software and Database Systems (July). IEEE, New York, pp. 163–169.

KRUEGER, P., AND FINKEL, R. A. 1983. An adaptive load balancing algorithm for a multicomputer. Unpublished manuscript, Computer Science Dept., Univ. of Wisconsin.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July), 558–565.

LAMPORT, L. 1984. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst. 6* (Apr.), 254–280.

LAMPSON, B. W. 1981. Atomic transactions. In *Distributed Systems—Architecture and Implementation*, B. W. Lampson, Ed. Springer-Verlag, Berlin and New York, pp. 246–265.

LAZOWSKA, E. D., LEVY, H. M., ALMES, G. T., FISCHER, M. J., FOWLER, R. J., AND VESTAL, S. C. 1981. The architecture of the Eden system. In *Proceedings of the 8th Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 14–16). ACM, New York, pp. 148–159.

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

LISKOV, B. 1982. On linguistic support for distributed programs. *IEEE Trans. Softw. Eng. SE-8* (May), 203–210.

LISKOV, B. 1984. Overview of the Argus language and system. Programming Methodology Group Memo 40. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., Feb.

LISKOV, B., AND SCHEIFLER, R. 1982. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst. 5*, 3 (July), 381–404. 1983. ACM, pp. 7–19, Jan. 1982.

LO, V. M. 1984. Heuristic algorithms for task assignment in distributed systems. In *Proceedings of the 4th International Conference on Distributed Computing Systems.* IEEE, New York, pp. 30–39.

LUDERER, G. W. R., CHE, H., HAGGERTY, J. P., KIRSLIS, P. A., AND MARSHALL, W. T. 1981. A distributed UNIX system based on a virtual circuit switch. In *Proceedings of the 8th Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 14–16). ACM, New York, pp. 160–168.

MAMRAK, S. A., MAURATH, P., GOMEZ, J., JANARDAN, S., AND NICHOLAS, C. 1982. Guest layering distributed processing support on local operating systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems.* IEEE, New York, pp. 854–859.

MARZULLO, K., AND OWICKI, S. 1985. Maintaining the time in a distributed system. *Oper. Syst. Rev. 19* (July), 44–54.

MENASCE, D., AND MUNTZ, R. 1979. Locking and deadlock detection in distributed databases. *IEEE Trans. Softw. Eng. SE-5* (May), 195–202.

MILLSTEIN, R. E. 1977. The national software works. In *Proceedings of the ACM Annual Conference* (Seattle, Wash., Oct. 16–19). ACM, New York, pp. 44–52.

MITCHELL, J. G., AND DION, J. 1982. A comparison of two network-based file servers. *Commun. ACM 25*, 4 (Apr.), 233–245.

MOHAN, C. K., AND WITTIE, L. D. 1985. Local reconfiguration of management trees in large networks. In *Proceedings of the 5th International Conference on Distributed Computing Systems* (May). IEEE, New York, pp. 386–393.

MULLENDER, S. J., AND TANENBAUM, A. S. 1984. Protection and resource control in distributed operating systems. *Comput. Networks 8* (Nov.), 421–432.

MULLENDER, S. J., AND TANENBAUM, A. S. 1985. A distributed file service based on optimistic concurrency control. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1–4). ACM, New York, pp. 51–62.

MULLENDER, S. J., AND TANENBAUM, A. S. 1986. The design of a capability-based distributed operating system. *Computer J.* (in press).

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

NELSON, B. J. 1981. Remote procedure call. Tech. Rep. CSL-81-9, Xerox Palo Alto Research Center, Palo Alto, Calif.

OBERMARCK, R. 1982. Distributed deadlock detection algorithm. *ACM Trans. Database Syst. 7* (June), 187–208.

OKI, B. M., LISKOV, B. H., AND SCHEIFLER, R. W. 1985. Reliable object storage to support atomic actions. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1–4). ACM, New York, pp. 147–159.

OUSTERHOUT, J. K. 1982. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems.* IEEE, New York, pp. 22–30.

PASHTAN, A. 1982. Object oriented operating systems: An emerging design methodology. In *Proceedings of the ACM National Conference* (Dallas, Tex., Oct. 25–27). ACM, New York, pp. 126–131.

POPEK, G., WALKER, B., CHOW, J., EDWARDS, D., KLINE, C., RUDISIN, G., AND THIEL, G. 1981. LOCUS: A network transparent, high reliability distributed system. In *Proceedings of the 8th Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 14–16). ACM, New York, pp. 160–168.

POWELL, M. L., AND MILLER, B. P. 1983. Process migration in DEMOS/MP. *Oper. Syst. Rev.* (ACM) *17*, 5, 110–119.

POWELL, M. L., AND PRESOTTO, D. L. 1983. Publishing—A reliable broadcast communication mechanism. *Oper. Syst. Rev.* (ACM) *17*, 5, 100–109.

PU, C., AND NOE, J. D. 1985. Nested transactions for general objects. Rep. TR-85-12-03, Computer Science Dept., Univ. of Washington, Seattle, Wash.

PU, C., NOE, J. D., AND PROUDFOOT, A. 1986. Regeneration of replicated objects: A technique and its Eden implementation. In *Proceedings of the 2nd International Conference on Data Engineering* (Los Angeles, Calif., Feb. 4–6). IEEE, New York, pp. 175–187.

RASHID, R. F., AND ROBERTSON, G. G. 1981. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 14–16). ACM, New York, pp. 64–75.

REED, D. P. 1983. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst. 1*, 1 (Feb.), 3–23.

REED, D. P., AND SVOBODOVA, L. 1981. 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.

REIF, J. H., AND SPIRAKIS, P. G. 1984. Real-time synchronization of interprocess communications. *ACM Trans. Program. Lang. Syst. 6*, 2 (Apr.), 215–238.

RITCHIE, D. M., AND THOMPSON, K. 1974. The UNIX time-sharing system. *Commun. ACM 19*, 7 (July), 365–375.

SALTZER, J. H., REED, D. P., AND CLARK, D. D. 1984. End-to-end arguments in system design. *ACM Trans. Comput. Syst. 2*, 4 (Nov.), 277–278.

SATYANARAYANAN, M., HOWARD, J., NICHOLS, D., SIDEBOTHAM, R., SPECTOR, A., AND WEST, M. 1985. The ITC distributed file system: Principles and design. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1–4). ACM, New York, pp. 35–50.

SCHROEDER, M., GIFFORD, D., AND NEEDHAM, R. 1985. A caching file system for a programmer's workstation. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1–4). ACM, New York, pp. 25–34.

SMITH, R. 1979. The contract net protocol: High-level communication and control in a distributed problem solver. In *Proceedings of the 1st International Conference on Distributed Computing Systems*. IEEE, New York, pp. 185–192.

SOLOMON, M. H., AND FINKEL, R. A. 1978. ROSCOE: A multimicrocomputer operating system. In *Proceedings of the 2nd Rocky Mountain Symposium on Microcomputers* (Aug.), pp. 201–210.

SOLOMON, M. H., AND FINKEL, R. A. 1979. The Roscoe distributed operating system. In *Proceedings of the 7th Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 10–12). ACM, New York, pp. 108–114.

SPECTOR, A. Z. 1982. Performing remote operations efficiently on a local computer network. *Commun. ACM 25*, 4 (Apr.), 246–260.

STANKOVIC, J. A., AND SIDHU, I. S. 1984. An adaptive bidding algorithm for processes, clusters, and distributed ups. In *Proceedings of the 4th International Conference on Distributed Computing Systems*. IEEE, New York, pp. 49–59.

STONE, H. S. 1977. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. Softw. Eng. SE-3* (Jan.), 88–93.

STONE, H. S. 1978. Critical load factors in distributed computer systems. *IEEE Trans. Softw. Eng. SE-4* (May), 254–258.

STONE, H. S., AND BOKHARI, S. H. 1978. Control of distributed processes. *Computer 11* (July), 97–106.

STONEBRAKER, M. 1981. Operating system support for database management. *Commun. ACM 24*, 7 (July), 412–418.

STURGIS, H. E., MITCHELL, J. G., AND ISRAEL, J. 1980. Issues in the design and use of a distributed file system. *Oper. Syst. Rev. 14* (July), 55–69.

SVENTEK, J., GREIMAN, W., O'DELL, M., AND JANSEN, A. 1983. Token ring local networks—A comparison of experimental and theoretical performance. Lawrence Berkeley Lab. Rep. 16254.

SVOBODOVA, L. 1981. A reliable object-oriented data repository for a distributed computer system. In *Proceedings of the 8th Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 14–16). ACM, New York, pp. 47–58.

SVOBODOVA, L. 1984. File servers for network-based distributed systems. *ACM Comput. Surv. 16*, 4 (Dec.), 353–398.

SWINEHART, D., MCDANIEL, G., AND BOGGS, D. 1979. WFS: A simple shared file system for a distributed environment. In *Proceedings of the 7th Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 10–12). ACM, New York, pp. 9–17.

TANENBAUM, A. S., AND MULLENDER, S. J. 1982. Operating system requirements for distributed data base systems. In *Distributed Data Bases*, H.-J. Schneider, Ed. North-Holland Publ., Amsterdam, pp. 105–114.

TANENBAUM, A. S., MULLENDER, S. J., AND VAN RENESSE, R. 1986. Using sparse capabilities in a distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computer Systems*. IEEE, New York, 1986, pp. 558–563.

VAN TILBORG, A. M., AND WITTIE, L. D. 1981. Wave scheduling: Distributed allocation of task forces in network computers. In *Proceedings of the 2nd International Conference on Distributed Computing Systems*. IEEE, New York, pp. 337–347.

WALKER, B., POPEK, G., ENGLISH, R., KLINE, C., AND THIEL, G. 1983. The LOCUS distributed operating system. *Oper. Syst. Rev.* (ACM) *17*, 5, 49–70.

WAMBECQ, A. 1983. NETIX: A network-using operating system, based on UNIX software. In *Proceedings of the* NFWO-ENRS Contact Group (Leuven, Belgium, Mar.).

WEINSTEIN, M. J., PAGE, T. W., JR., LIVESEY, B. K., AND POPEK, G. J. 1985. Transactions and synchronization in a distributed operating system. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1–4). ACM, New York, pp. 115–125.

WILKES, M. V., AND NEEDHAM, R. M. 1980. The Cambridge model distributed system. *Oper. Syst. Rev. 14* (Jan.), 21–29.

WITTIE, L., AND CURTIS, R. 1985. Time management for debugging distributed systems. In *Proceedings of the 5th International Conference on Distributed Computing Systems* (May). IEEE, New York, pp. 549–551.

WITTIE, L. D., AND VAN TILBORG, A. M. 1980. MICROS, a distributed operating system for MICRONET, a reconfigurable network computer. *IEEE Trans. Comput C-29* (Dec.), 1133–1144.

WUPIT, A. 1983. *Comparison of UNIX network systems.* ACM, New York, pp. 99–108.

ZIMMERMANN, H. 1980. OSI reference model—The ISO model of architecture for open systems interconnection. *IEEE Trans. Commun. COM-28* (Apr.), 425–432.

ZIMMERMANN, H., BANINO, J.-S., CARISTAN, A., GUILLEMONT, M., AND MORISSET, G. 1981. Basic concepts for the support of distributed systems: The chorus approach. In *Proceedings of the 2nd International Conference on Distributed Computing Systems.* IEEE, New York, pp. 60–66.

Received July 1985; final revision accepted April 1986.