

Beyond UNIX—A True Distributed System for the 1990s

Andrew S. Tanenbaum

Robbert van Renesse

Hans van Staveren

Gregory J. Sharp

Dept. of Mathematics and Computer Science

Vrije Universiteit

De Boelelaan 1081

1081 HV Amsterdam, The Netherlands

Internet: ast@cs.vu.nl, cogito@cs.vu.nl, sater@cs.vu.nl, gregor@cs.vu.nl

ABSTRACT

UNIX has been around now for almost 20 years. At the time UNIX began, most departments felt themselves well-endowed indeed if they owned a single PDP-11/45 with 256K memory and a 2.5M RK05 disk. Nowadays a laptop would be embarrassed to have only that. It is our hypothesis that UNIX is no longer the appropriate kind of operating system for the 1990s. In this paper, a new system, Amoeba, will be described, that we believe meets the requirements for distributed computing in the 1990s.

1. Introduction

UNIX is now almost 20 years old. Although it has gotten much bigger over the years, the basic ideas have not really changed since it was created in the early 1970s. Furthermore, many of the ideas in UNIX actually go back to MULTICS, which was designed in the early 1960s. As a result, UNIX may no longer be the ideal operating system for the 1990s and beyond. Perhaps it is time to start over fresh with something new. In this paper we describe the Amoeba distributed operating system, which has been designed and implemented with the technology of the 1990s in mind [van Renesse et al., 1989a, 1989b; Tanenbaum et al., 1986]. We believe that Amoeba is a worthy successor to UNIX.

What are the key characteristics of computing now and in the future? We are convinced that two factors will dominate the next decade:

- The need for physically distributed hardware
- The need for logically centralized software

Let us now discuss these in turn.

First, computers are becoming cheaper at an enormous rate. In the 1970s, it was normal for many people to share a single mainframe or minicomputer by running a timesharing system on it. Each user had a terminal with which to access the computer. The ratio of computers to people was very low, often 20 or 50 or even 100 people per machine.

In the 1980s, the personal computer and personal workstation became popular. By the end of the decade, many universities and companies operated using a model in which each person had his or her own machine, all connected by a local area network. The ratio of computers to people became approximately 1 to 1, as many machines as people.

In the 1990s, this trend will continue. We will soon come to a situation in which it is economically feasible to have 20 or 50 or even 100 computers per person. Clearly the current model of giving each person a personal computer or workstation breaks down under these conditions. Nevertheless, the availability of large numbers of powerful single-chip processors is a given. Any system for the 1990s must address itself to the issue of how to deal with a system containing hundreds, if not thousands, of processors, very likely distributed over a considerable area.

The second factor mentioned above is the need for logically centralized software. While it is currently possible to physically connect up a few dozen machines on a local area network, the result is often unpleasant for the users. In many ways, the 1970s model of having one machine that everyone used was in fact much simpler and easier to use than the current personal computer model. With only a little effort, giving each user dozens of computers could produce a complete disaster.

We believe what users want is a system built out of large numbers of powerful microprocessors that take advantage of the current hardware technology, but which together act in a coherent way that is as easy to use and understand as an old PDP-11 timesharing system. Users do not want, and rarely fully understand concepts such as remote mounting, yellow pages, and similar bizarre and complicated things.

We must produce a new generation of operating systems that tie all the pieces together and make the collection of hardware boxes look like a single, integrated machine, rather than a bunch of distinct machines that communicate using some form of network protocol. The user logging into the system should not be aware of how many machines there are, where they are located, what their functions are, where the files are (or how many copies there are), how many processors are needed to run any particular job, or anything else about the physical distribution of the hardware. This is the challenge of the 1990s.

2. Overview of Amoeba

As a first step towards a completely new operating system designed expressly to meet these goals, we began the Amoeba project at the Vrije Universiteit in Amsterdam. Subsequently, the Vrije Universiteit has teamed up with the Centrum voor Wiskunde en Informatica to continue developing Amoeba jointly. In this paper we will describe the Amoeba system and try to show why we think it is appropriate for the coming decade.

Before describing the software, it is worth saying something about the system architecture on which Amoeba runs. The Amoeba architecture consists of four principal components, as shown in Fig. 1.

- Workstations with a window system for providing user access
- Pool processors for computing
- Specialized servers such as file servers and directory servers
- Gateways to other systems

The key idea here is that the workstations are basically terminals. A typical workstation might be a Sun-3 or an X-terminal. The job of the workstation is to run the window manager and interact with the user via the keyboard and mouse. With few exceptions, such as an editor, heavy computing generally does not occur on the workstations. They are really just glorified terminals.

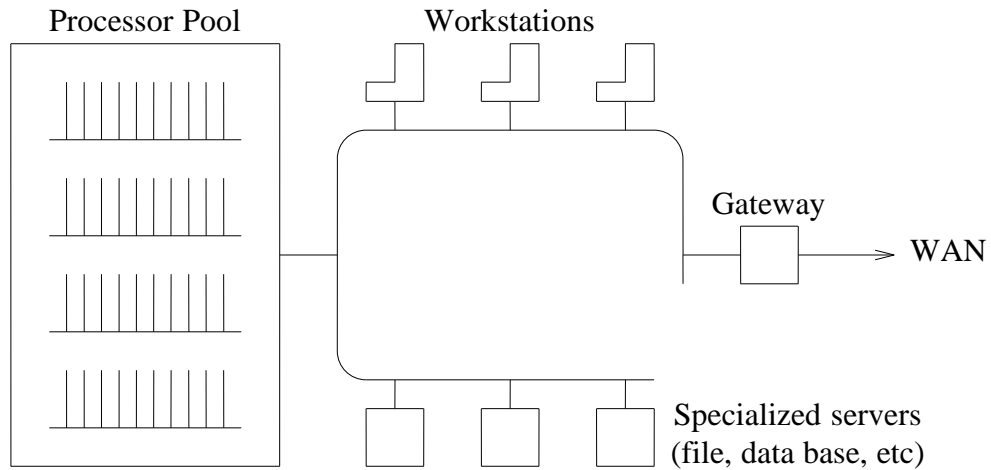


Fig. 1. The Amoeba architecture.

The computing power lies in the pool processors. In our installation, this consists of a several standard 19-inch equipment racks containing a total of 48 single board computers, each having a powerful CPU (68020 and 68030 at present), 2-4M of memory, and a network connection (Ethernet). When a job needs computing power, it asks the process server to temporarily allocate it some number of processors, which it then uses and then returns. A typical use might be running the *make* program. Suppose *make* discovers that it needs to do 8 compilations, and the compiler has 5 passes. Then 40 processors could be allocated (if available) and all the passes of all the compilations could proceed in parallel. As soon as they were finished, the processors would go back into the pool to be available for another request by another user.

The specialized servers are machines that need dedicated resources of some kind. The file server runs best on a machine with disks, for example.

Finally, the gateways are used to connect up multiple Amoeba systems at different sites in different cities or even different countries. Their job is to protect the local machines from the wide-area protocols, to make it possible to access a machine in a different city without having to even know that it is distant. The gateways handle all the protocol wrapping and unwrapping transparently.

The Amoeba software is object-based. The system can be viewed as a collection of objects, on each of which there is a set of operations that can be performed. For a file object, for example, typical operations are reading, writing, appending, and deleting. The list of allowed operations is defined by the person who designs the object and who writes the code to implement it. Both hardware and software objects exist.

Associated with each object is a *capability*, a kind of ticket or key that allows the holder of the capability to perform some (not necessarily all) operations on that object. A user process might, for example, have a capability for a file that permitted it to read the file, but not to modify it. Capabilities are protected cryptographically to prevent users from tampering with them.

Each user process owns some collection of capabilities, which together define the set of objects it may access and the type of operations he may perform on each. Thus capabilities provide a unified mechanism for naming, accessing, and protecting objects. From the user's perspective, the function of the operating system is to create an environment in which objects can be created and manipulated in a protected way.

This object-based model visible to the users is implemented using remote procedure call (Birrell and Nelson, 1984; Tanenbaum and van Renesse, 1988). Associated with each object is a *server* process that manages the object. When a user process wants to perform an operation

on an object, it sends a request message to the server that manages the object. The message contains the capability for the object, a specification of the operation to be performed, and any parameters the operation requires. The user, known as the *client*, then blocks. After the server has performed the operation, it sends back a reply message that unblocks the client. The combination of sending a request message, blocking, and accepting a reply message forms the remote procedure call, which can be encapsulated using stub routines, to make the entire remote operation look like a local procedure call.

The Amoeba kernel basically handles communication and some process management, and little else. The kernel takes care of sending and receiving messages, scheduling processes, and some low-level memory management. Everything else is done by user-level server processes.

3. Naming and Protection of Objects

Amoeba has a unified scheme for doing naming and protection in a location independent way. The system can be viewed as a collection of objects, on each of which there is a set of operations that can be performed. For a file object, for example, typical operations are reading, writing, appending, and deleting. The list of allowed operations is defined by the person who designs the object and who writes the code to implement it. Both hardware and software objects exist.

As described above, associated with each object is a *capability* [Dennis and Van Horn, 1966] that controls access to the object. The structure of a capability is shown in Fig. 2. It is 128 bits long and contains four fields. The first field is the *server port*, and is used to identify the (server) process that manages the object. It is in effect a 48-bit random number chosen by the server.

48	24	8	48
Server port	Object number	Rights	Check field

Fig. 2. A capability. The numbers are the current sizes in bits.

The second field is the *object number*, which is used by the server to identify which of its objects is being addressed. Together, the server port and object number uniquely identify the object on which the operation is to be performed.

The third field is the *rights* field, which contains a bit map telling which operations the holder of the capability may perform. If all the bits are 1s, all operations are allowed. However, if some of the bits are 0s, the holder of the capability may not perform the corresponding operations.

To prevent users from just turning all the 0 bits in the rights field into 1 bits, a cryptographic protection scheme is used. When a server is asked to create an object, it picks an available slot in its internal tables, puts the information about the object in there along with a newly generated 48-bit random number. The index into the table is put into the object number field of the capability, the rights bits are all set to 1, and the newly-generated random number is put into the *check field* of the capability. This is an *owner capability*, and can be used to perform all operations on the object.

The owner can construct a new capability with a subset of the rights by turning off some of the rights bits and then XOR-ing the rights field with the random number in the check field. The result of this operation is then run through a (publicly-known) *one-way function* to produce a new 48-bit number that is put in the check field of the new capability.

The key property required of the one-way function, f , is that given the original 48-bit

number, N (from the owner capability) and the unencrypted rights field, R , it is easy to compute $C = f(N \text{ XOR } R)$, but given only C it is nearly impossible to find an argument to f that produces the given C . Such functions are known [Evans et al., 1974].

When a capability arrives at a server, the server uses the object field to index into its tables to locate the information about the object. It then checks to see if all the rights bits are on. If so, the server knows that the capability is (or is claimed to be) an owner capability, so it just compares the original random number in its table with the contents of the check field. If they agree, the capability is considered valid and the desired operation is performed.

If some of the rights bits are 0, the server knows that it is dealing with a derived capability, so it performs an XOR of the original random number in its table with the rights field of the capability. This number is then run through the one-way function. If the output of the one-way function agrees with the contents of the check field, the capability is deemed valid, and the requested operation is performed if its rights bit is set to 1. Due to the fact that the one-way function cannot be inverted, it is not possible for a user to decrypt a capability to get the original random number in order to generate a false capability with more rights.

4. Operations on Objects

Three primitives are provided for performing operations on objects. Client processes use

do_operation(req_header, req_buffer, req_size, rep_header, rep_buffer, rep_size)

to ask a server to perform a remote operation. The first three parameters are used to specify the request, and are analogous to the parameters to the UNIX *write(fd, buf, count)* system call. The last three are used to receive the reply, and are analogous to the parameters of the UNIX *read(fd, buf, count)* system call. It is as though a process writes a command to the server, then immediately does a read for the reply, blocking until the reply comes in.

The other two primitives are used by server processes.

get_request(req_header, req_buffer, req_size)

is used to block waiting for a request message to come in. When it does, the message is processed and the reply is sent back using

put_reply(rep_header, rep_buffer, rep_size)

Thus a typical server consists of a loop getting requests, carrying them out, and sending replies. Most of the time, most servers are blocked waiting for the next request.

Amoeba supports multiple *threads* within a process. The threads run in pseudoparallel and all occupy the same address space. Threads can synchronize using semaphore operations. For example, a file server could be constructed with multiple threads. When a request came in, one thread would accept it and start working on it. If this blocked waiting for a disk block, another thread could continue processing another request. Nevertheless, all the threads could share a single buffer space directly accessible to all of them.

5. The Amoeba File System

The Amoeba file system consists of two parts, the Bullet File Service and the SOAP Directory Service. The Bullet Service handles the actual storage of information, while the Directory handles the user interface to it. Let us now look at them in turn.

The Bullet Service is a highly unusual file service, which may have one or more servers providing it. Each of the Bullet Servers support only three principal operations:

- *read_file*

- create_file
- delete_file

When a file is created, the user normally provides all the data at once, creating the file and getting back a capability for it. This capability can later be used to read the data back or delete the file.

All files are *immutable*, that is, once created they cannot be changed. Notice that there is no *write* operation supported. The reason for having immutable files relates to replication and caching, and will be discussed below.

Files are stored contiguously, both on the disk and in Bullet Servers' caches, as illustrated in Fig. 3. The administrative information for a file is then reduced to its origin and size plus some ownership data. The complete administrative table is loaded into the Bullet Server's memory when it is booted. When a *read* operation is done, the object number in the capability is used as an index into this table, and the file is read into the cache in a single (possibly multitrack) disk operation.

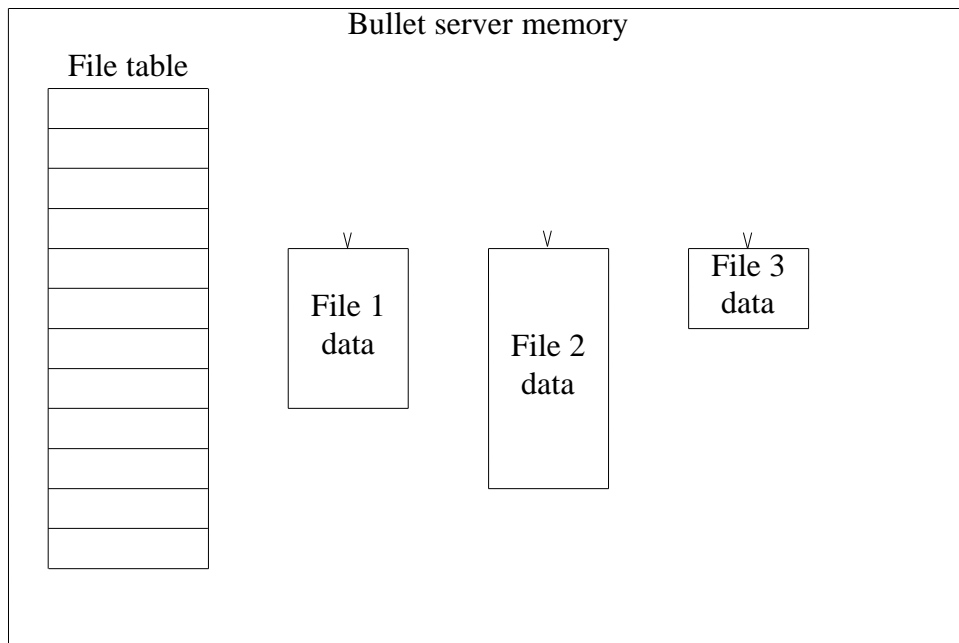


Fig. 3. Bullet Server file representation.

Although the Bullet Service wastes some space due to fragmentation, its use of contiguous files achieves an enormous performance that easily compensates for having to buy an 800M disk to store, say, 500M worth of data.

When a process creates a file and gets back a capability, it will normally want to give the file a symbolic name and make a directory entry for it. This is achieved by using the Directory Service, which maps symbolic names onto capabilities. To a first approximation, a directory is a set of (name, capability) pairs. The basic operations on directory objects are:

- lookup
- enter
- delete

The first one looks up a symbolic object name in a directory and returns its capability. The other two enter and delete objects from directories. Since directories themselves are objects, a directory may contain capabilities for other directories, thus potentially allowing users to build an arbitrary graph structure.

Complex sharing can be achieved by making directories more sophisticated than we have just described. Conceptually, a directory consists of multiple columns, with the name in column 0 and capabilities in the other columns. Each column may have different access rights. For example, column 1 might be for the owner, and have read and write access to all files, while column 2 might be for other people with read-only access. For objects that support more kinds of operations, more columns could be provided for other access classes. Simulating the UNIX rwx bits for the owner, group, and other is straightforward to implement using three columns.

6. Reliable Broadcasting

Many distributed applications involve replicated data in one form or another. For this reason, we have been looking at including broadcasting in Amoeba as an efficient way to support it. We now believe we understand the problem well enough to include the necessary primitives in the kernel, starting with the next version. Below we describe the basic ideas involved.

The key concept is *reliable broadcasting*. By this we mean that a process can do a broadcast operation and not have to worry about the possibility of messages being lost or duplicated. It can just assume that all messages will be correctly delivered, in the proper sequence. It is up to the implementation to achieve this goal in the face of unreliable hardware and lost messages. A rough analogy can be made with error-correcting memories. The user of a machine with an error-correcting memory does not have to explicitly program a routine to handle detected errors. That is somebody else's problem.

What the Amoeba kernel must support is the basic broadcasting mechanism. The way it does this is to offer a primitive that allows a process to issue a broadcast. This primitive is *implemented* by sending a point-to-point message to a special server called the *sequencer*, which runs on the same hardware as all the others, and can be easily replaced if it should fail. The sequencer assigns the first unused sequence number to the message and broadcasts it on the LAN.

Most of the machines will get this broadcast, verify that the sequence number is correct, and process it. However, a small number may miss it due to communication errors, lack of buffer space, or other reasons. When the next broadcast comes in, the gap in the sequence space will be noticed, and the missing message can be fetched from the sequencer, which stores old ones. To avoid forcing the sequencer to maintain the entire history, each message sent contains an acknowledgement of the last broadcast received. When the sequencer discovers that everybody has seen all messages up through n , it can delete messages 0 through n from its history buffer.

The actual protocols used are described in [Kaashoek and Tanenbaum, 1989]. The details are not important here. The main thing to remember is that user programs can use this facility to do reliable broadcasting without having to worry about what happens if something goes wrong. This division of labour greatly simplifies parallel programming, as will be discussed later in this paper.

7. Amoeba on Wide-Area Networks

Although the initial goal was to produce a distributed system that worked on a single LAN, we soon became involved in a COST-11 project involving people from several countries. The question arose: "Could we extend Amoeba to a wide-area network without compromising the performance of the local case?" In particular, we considered, but quickly rejected the possibility of using TCP/IP, X.25, or ISO OSI as the base for the local communication instead of RPC. There would be far too much loss of performance.

Instead we adopted a solution involving transparent gateways, as shown in Fig. 4. The basic

idea is to divide the world up into *domains*. A domain has the property that it is possible and efficient to broadcast messages from any machine to all the other machines within a domain. A domain might be a single LAN, or possibly a group of LANs interconnected by repeaters and bridges. Broadcasting is important because port location is done by the kernel using broadcasting. When a user does a *do_operation* call, the kernel looks up the port in its cache. If the port is not there, it broadcasts a locate request for it. Clearly this will not work over a wide-area network.

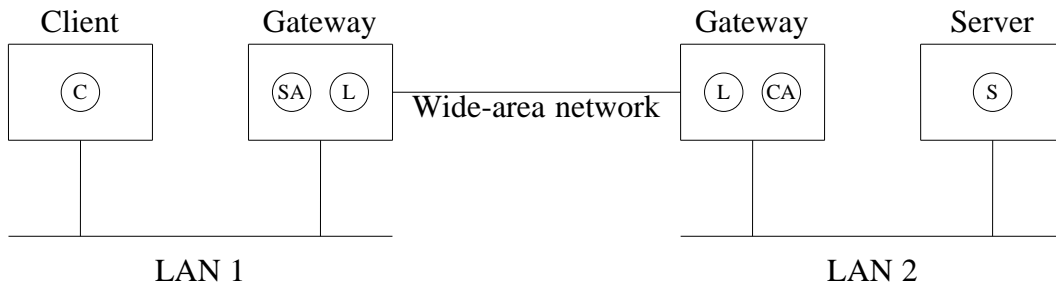


Fig. 4. Wide-area communication in Amoeba involves six processes.

Instead a different solution is used, called *publishing*. With this solution, each domain has a gateway to the remote domains. When a service wants to make itself known outside its own domain, it publishes its port by sending a (point-to-point) message to the gateway of each domain in which it wants its service known. A laser printer service in the physics building at a university in California may or may not want to be known in the engineering building across campus, but almost certainly does not want to be known in New York.

When a gateway receives a message announcing a service, it makes an entry in its tables recording the port being used and other information. Now consider what happens when a process tries to use a remote service. It does an RPC, as usual. The kernel on its machine looks the port up in its cache and does not find it, so it does a broadcast locally. The gateway sees the broadcast and sends back a reply saying that RPCs for that port should be directed to it.

When the RPC comes in to the gateway machine shortly thereafter, a *server agent* is created on the gateway machine. The server agent takes the RPC and forwards it to a *link* process on the same machine. The link process knows about the wide-area networks and their protocols, and forwards the message to its peer link agent on the destination machine, where a *client agent* is created. The client agent then proceeds to do an RPC to the server.

The beauty of this scheme is that it is completely transparent. The client does not know it is using a remote service. It does RPCs to a local process (the server agent) and everything works the way it normally does. The client does not know (or have to know) that the server agent really is not doing the work itself. Similarly, the server gets an RPC from a local process, the client agent, and sends a reply back to this local process. That the client agent is in fact working on behalf of a third party is of no concern to it.

Of course the two agents are aware that wide-area communication is taking place, but that is their job. Furthermore, they are generated on-the-fly as needed, so they are of no concern to anyone.

The only two processes that are aware of the wide-area network protocols are the two link processes. Their function is to isolate these protocols from everything else in the system. The link processes can use X.25 or TCP/IP, or OSI or whatever they want to, without other processes being aware. In fact, it is easy to have multiple wide-area networks in use at once, each one with its own link processes.

Finally, it is worth pointing out explicitly that this arrangement solves the problem that we originally posed—how to add wide-area to Amoeba without affecting local communication.

Normal RPC between clients and servers on a single LAN is in no way affected by the presence or absence of a gateway on their network. They use the same protocols and procedures in all cases, and the local RPCs are just as fast as they would have been had no effort been spent on wide-area at all.

8. Performance of Amoeba

Amoeba was designed to be very fast. In fact, that was probably its second most important goal (after transparency). In this section we will look at some of the initial performance figures based on measurements we have made using 16 MHz 68020 processors (Tadpole VME single board computers) communicating over an Ethernet. For comparison purposes, we also measured SUN OS 3.5 RPC running on Sun 3/50 workstations, which are about the same speed as the Tadpole boards.

Two cases are of interest: the delay for the case of short messages and the bandwidth for the case of long messages. The former is important for quick response to interactive commands; the latter is important to moving large volumes of data around. We made measurements for 4 byte, 8 Kbyte, and 30 Kbyte messages for Amoeba, and 4 byte and 8Kbyte messages for SUN RPC. A message of 30K is impossible on SUN RPC.

In all cases, we measured the end-to-end performance, that is, the time for a client running in user mode to do its RPC, trap to the kernel, have the kernel send the message over the Ethernet to the remote kernel, where it was passed up to the server running in user mode, and then all the way back again to the client in user mode.

	<i>Delay (msec)</i>			<i>Bandwidth (Kbytes/sec)</i>		
	case 1 <i>(4 bytes)</i>	case 2 <i>(8 Kb)</i>	case 3 <i>(30 Kb)</i>	case 1 <i>(4 bytes)</i>	case 2 <i>(8 Kb)</i>	case 3 <i>(30 Kb)</i>
Amoeba	1.4	13.1	44.0	2.9	625	677
SUN	12.2	40.6	imposs.	0.3	202	imposs.

(a) (b)

Fig. 5. Amoeba and SUN OS performance.

From Fig. 5 we see that an RPC of 4 bytes takes 1.4 msec for the request to be sent and the reply to come back. For SUN RPC, the comparable figure is 12.2 msec, or 8 times slower. Similarly, the bandwidth achieved for 8K RPCs is 625 Kbytes/sec for Amoeba and 202 for SUN RPC. Our initial finding is that Amoeba RPC is 3 to 8 times faster than SUN RPC.

9. Applications

The preceding sections have discussed the Amoeba kernel and some of the key servers that can be thought of as part of the system, even though they run in user mode. Below we will look at some applications that run on Amoeba.

9.1. UNIX Emulation

For various reasons, it was thought wise to include some kind of UNIX emulation facility. Since the whole design of Amoeba is so different from UNIX, it would be very difficult, if not impossible, to provide binary compatibility. Instead, we opted for making a special library containing routines for most of the common UNIX system calls. These routines use the Bullet Service, Directory Service, and others to get the job done, but to the user, look like ordinary UNIX calls. This approach works best for the file system calls.

To emulate *fork*, *exec*, *signal*, and *kill*, a library was not enough. Instead, a *session server* was written, which handles these calls and otherwise manages state information in an otherwise largely stateless system. The more exotic calls have not been implemented. Our goal is to ultimately implement all of the IEEE POSIX P1003.1 standard.

About 150 utilities have been ported to Amoeba. Some have been taken from MINIX, some have been written from scratch. Many of them use the UNIX emulation package, which is called *Ajax*. None of them contain any AT&T code.

In addition, X windows has been ported to Amoeba, and a TCP/IP server has been written to permit communication with the outside world.

9.2. Make

Another important application is a new, enhanced version of *make*, called *amake* [Baalbergen et al., 1989], that runs its compilations in parallel. When trying to make a target, such as *a.out*, *amake* checks to see if all the necessary steps can be run in parallel. For example, compilations to produce the necessary *.o* files can often be run simultaneously. If possible, they are run at the same time on different pool processors. However, if one step cannot proceed because it needs the results of a previous step, then it is postponed until later.

Since it was necessary to redesign *make* anyway to make it work in parallel, it was completely rethought from scratch to solve a number of inherent problems in the design of the original.

Unlike *make*, *amake* does not use *Makefiles* containing dependency lists. Instead, the compilers and other programs have been modified slightly to report back after completion which files were included. All this information goes into a mini data base maintained by *amake*. The next time *amake* is called to make a given target, it checks this data base to see what the dependencies are. If none of the dependents of a target have changed, the target is still valid and no work need be done.

Amake has a more general idea of dependency than *make*. For example, if a new compiler has been installed in */bin*, or different flags are used, these will be detected and recompilation forced where necessary. *Amake* will happily provide the user with the complete dependency graph if asked. Thus instead of the user providing the dependency information to *make*, (*a*)*make* figures out everything itself. provides it to the user. In practice, not having to deal with *Makefiles* (not even automatically generated ones), has proven very popular with the users.

Determining whether a file has changed or not is not entirely trivial on a distributed system such as Amoeba. One cannot assume that different machines (e.g., different file servers) have their clocks synchronized. If one were to rely on the time of last modification, race conditions could arise.

Instead, use is made of the fact that the Bullet Service supports only immutable files. Suppose that *amake* discovers that *file.c* depends on *file.h*. It records this fact in its data base, along with the capabilities for *file.o*, *file.c*, and *file.h*. If *amake* is invoked later, it can tell if *file.o* has to be regenerated by looking up the capabilities for *file.h* and *file.c* using the directory service and comparing them to the values in the data base. If they are the same, the original files are still valid. (Remember that a Bullet file cannot be changed. If the file is edited, a new version, with a new capability is created, and that capability is stored in the directory, replacing the old one, which is then deleted.)

9.3. Parallel Programming

Since Amoeba supports parallel as well as distributed programming, we have designed a language, *Orca*, in which to program it [Bal, 1990]. The basic model used by Orca is that of abstract data types that are shared among processes running on multiple machines. Each abstract data type has operations that may be performed on it. Any process can perform these operations at any time, and they will be carried out automatically on all the machines. The language guarantees that no matter how many processes attempt to execute operations simultaneously, each operation will be performed atomically, without interference by other processes.

Several implementations of the run-time system are available. The most efficient one uses the reliable broadcasting described in Sec. 6. Every machine maintains a copy of all the abstract data types locally. Read operations can be carried out locally, without any network traffic. These go at full speed. Write operations use the reliable broadcast to simultaneously update all copies at once. Since the broadcasting algorithm guarantees that all copies are updated in the same order, race conditions are avoided. In this way we have a simple, elegant, and efficient technique for distributed and parallel computation.

10. Conclusions

In our view, the key research issue for the 1990s is how to deal with the enormous amount of computing power that will become available during the coming decade. The workstation model now in vogue will become increasingly unattractive as the ratio of CPUs to people becomes 10 or more. Giving everyone a medium-sized, dedicated multiprocessor is undesirable because most of the potential will be wasted, but worse yet, when a user really needs the total computing power, it will be unavailable.

Our solution is to give each user a limited amount of computing power in the form of a personal workstation, with a large shared processor pool for the main compute load. This will be accompanied by a fast file server using contiguous files protected by location-independent capabilities. Access to remotes sites is done in a transparent way. Finally, a language and run-time system have been devised to make parallel and distributed programming much easier than has been the case until now.

11. References

- Baalbergen, E.H., Verstoep, K., and Tanenbaum, A.S.: "On the Design of the Amoeba Configuration Manager," *ACM SIGSOFT Software Engineering Notes*, vol. 17, Nov. 1989 (*Proc. 2nd Int'l Workshop on Software Configuration Management*) ACM, 1989.
- Bal, H.E., Kaashoek, M.F., and Tanenbaum, A.S.: "Experience with Distributed Programming in Orca," *Proc. Int'l Conf. on Comp. Languages '90*, IEEE, 1990.
- Birrell, A.D., and Nelson, B.J.: "Implementing Remote Procedure Calls," *ACM Trans. Comput. Systems*, vol. 2, pp. 39-59, Feb. 1984.
- Dennis, J.B. and Van Horn, E.C.: "Programming Semantics for Multiprogrammed Computations," *Commun. ACM*, vol. 9, pp. 143-154, March 1966.
- Evans, A., Kantrowitz, W., and Weiss, E.: "A User Authentication Scheme not Requiring Security in the Computer," *Commun. ACM*, vol. 17, pp. 437-442, Aug. 1974.
- Kaashoek, M.F., Tanenbaum, A.S., Flynn Hummel, S., and Bal, H.E.: "An Efficient Reliable

Broadcast Protocol," *Operating Systems Review*, vol. 23, pp. 5-19, Oct. 1989.

Renesse, R. van, Tanenbaum, A.S., and Wilschut, A: "The Design of a High-Performance File Server" *Proc. Ninth Int'l Conf. on Distr. Comp. Systems*, IEEE, pp. 22-27, 1989a.

Renesse, R. van, Staveren, H. van, and Tanenbaum, A.S.: "Performance of the Amoeba Distributed Operating System," *Software—Practice and Experience*, vol. 19, pp. 223-234, March 1989b.

Tanenbaum, A.S., and Renesse, R. van: "A Critique of the Remote Procedure Call Paradigm" *Proc. Euteco '88* pp. 775-783, 1988.

Tanenbaum, A.S., Mullender, S.J., and van Renesse, R.: "Using Sparse Capabilities in a Distributed Operating System" *Proc. Sixth International Conf. on Distr. Computer Systems*, IEEE, pp. 558-563, 1986.