THE AMOEBA DISTRIBUTED OPERATING SYSTEM—A STATUS REPORT

Andrew S. Tanenbaum
M. Frans Kaashoek
Robbert van Renesse
Henri E. Bal

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands

ABSTRACT

As the price of CPU chips continues to fall rapidly, it will soon be economically feasible to build computer systems containing a large number of processors. The question of how this computing power should be organized, and what kind of operating system is appropriate then arises. Our research during the past decade has focused on these issues and led to the design of a distributed operating system, called Amoeba, that is intended for systems with large numbers of computers. In this paper we describe Amoeba, its philosophy, its design, its applications, and some experience with it.

1. INTRODUCTION

The cost of CPU chips is expected to continue declining during the coming decade, leading to systems containing a large number of processors. Connecting all these processors using standard technology (e.g., a LAN) is easy. The hard part is designing and implementing software to manage and use all of this computing power in a convenient way. In this paper we describe a distributed operating system that we have written, called Amoeba (Mullender et al., 1990; Tanenbaum et al., 1990; Van Renesse et al., 1989), that we believe is appropriate for the distributed systems of the 1990s.

One basic idea underlies Amoeba: to the user, the complete system should look like a single computer. By this we mean that the current model in which networked computers can only share resources with considerable difficulty will have to be replaced by a model in which the complete collection of hardware appears to the user to be a traditional uniprocessor timesharing system. Users should not be aware of which machines they are using, or how many machines they are using, or where these machines are located. Nor should they be aware of where their files are stored or how many copies are being maintained or what replication algorithms are being used. To the users, there should be a single integrated system which they deal with. It is the job of the operating system and compilers to provide the user with the illusion of a single system, while at the same time efficiently managing the resources necessary to support this illusion.

This is the challenge we have set for ourselves in designing Amoeba. Although we are by no means finished, and considerable research is still needed, in this paper we

present a status report of Amoeba and our thoughts about how it is likely to evolve.

The outline of this paper is as follows. In Sec. 2 we will discuss the type of hardware configuration for which Amoeba has been designed. In Sec. 3 we will begin our discussion of Amoeba itself, starting with the microkernel. Since much of the traditional operating system functionality is outside the microkernel, running as server processes, in Sec. 4 we will discuss some of these servers. Then in Sec. 5, we will discuss a few applications of Amoeba to date. In Sec. 6 we take a brief look at how Amoeba can be used over wide-area networks. In Sec. 7 we discuss our experiences with Amoeba, both good and bad. Sections 8 and 9 compare Amoeba to other systems and summarize our conclusions, respectively.

2. THE AMOEBA MODEL

Before describing how Amoeba is structured, it is useful to first outline the kind of hardware configuration for which Amoeba is designed, since it differs somewhat from what most organizations presently have. The driving force behind the system architecture is the need to incorporate large numbers of CPUs in a straightforward way. In other words, what do you do when you can afford 10 or 100 CPUs per user? One solution is to give each user a personal 10-node or 100-node multiprocessor. However, we do not believe this is an effective way to spend the available budget. Most of the time, nearly all the processors will be idle, which by itself is not so bad. However, some users will want to run massively parallel programs, and will not be able to harness all the idle CPU cycles because they are in other users' personal machines.

Instead of this personal multiprocessor approach, we believe that a better model is shown in Fig. 1. In this model, all the computing power is located in one or more *processor pools*. Each processor pool consists of a substantial number of CPUs, each with its own local memory and its own network connection. At present, we have a prototype system operational, consisting of three standard 19-inch equipment racks, each holding 16 single board computers (68020 and 68030, with 3-4M RAM per CPU). Each CPU has its own Ethernet connection; shared memory is not present. While it would be easy to build a 16-node shared memory multiprocessor, it would not be easy to build a 1000-node shared memory multiprocessor. Thus our model does not assume that any of the CPUs share physical memory, in order to make it possible to scale the system. However, if shared memory is present, it can be utilized to optimize message passing by just doing memory-to-memory copying instead of sending messages over the LAN.

Pool processors are not "owned" by any one user. When a user types a command, the system automatically and dynamically allocates one or more processors for that command. When the command completes, the processors are released and go back into the pool, waiting for the next command, very likely from a different user. If there is a shortage of pool processors, individual processors are timeshared, with new jobs being assigned to the most lightly loaded CPUs. The important point to note here is that this model is quite different from current systems in which each user has exactly one personal workstation for all his computing activities. The pool processor model is more flexible, and provides for a better sharing of resources.

The second element in our architecture is the workstation. It is through the workstation that the user accesses the system. Although Amoeba does not forbid running user

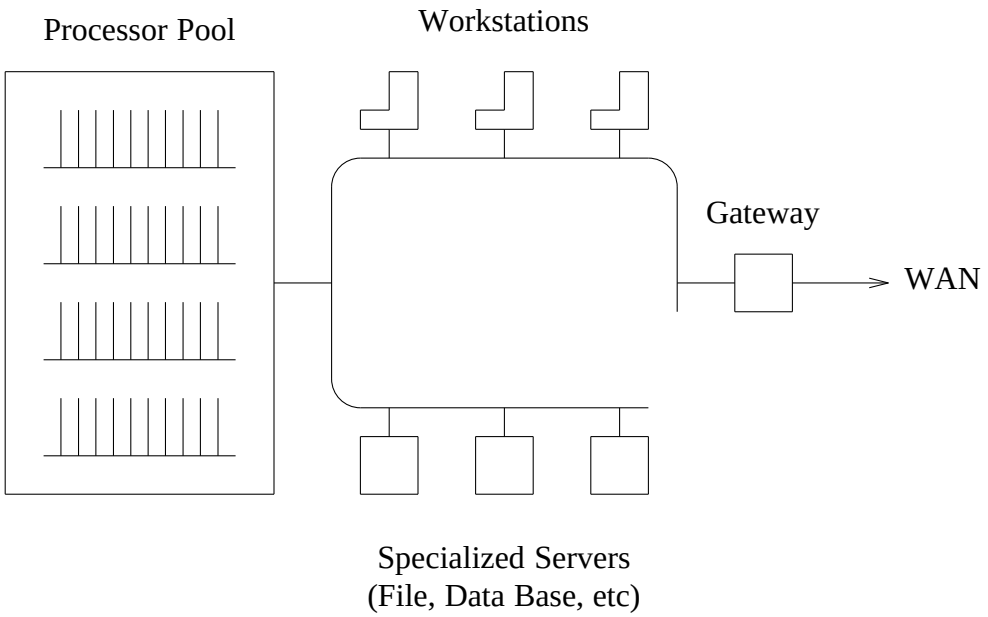

Fig. 1. The Amoeba System Architecture.

programs on the workstation, normally the only program that runs there is the window manager. For this reason, X-terminals can also be used as workstations.

Another important component of the Amoeba configuration consists of specialized servers, such as file servers, which for hardware or software reasons need to run on a separate processor. Finally, we have the gateway, which interfaces to wide-area networks and isolates Amoeba from the protocols and idiosyncracies of the wide-area networks in a transparent way.

3. THE AMOEBA MICROKERNEL

Now we come to the structure of the Amoeba software. The software consists of two basic pieces: a microkernel, which runs on every processor, and a collection of servers that provide most of the traditional operating system functionality. In this section we will describe the microkernel. In the next one we will describe some servers.

The Amoeba microkernel runs on all machines in the system. It has four primary functions:

1. Manage processes and threads within these processes.
2. Provide low-level memory management support.
3. Support transparent communication between arbitrary threads.
4. Handle I/O.

Let us consider each of these in turn.

Like most operating systems, Amoeba supports the concept of a process. In

addition, Amoeba also supports multiple threads of control, or just *threads* for short, within a single address space. A process with one thread is essentially the same as a process in UNIX.† Such a process has a single address space, a set of registers, a program counter, and a stack. Threads are illustrated in Fig. 2.

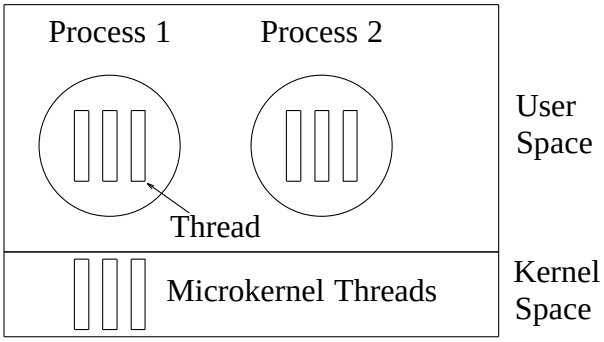

Fig. 2. Threads in Amoeba.

In contrast, although a process with multiple threads still has a single address space shared by all threads, each thread logically has its own registers, its own program counter, and its own stack. In effect, a collection of threads in a process is similar to a collection of independent processes in UNIX, with the one exception that they all share a single common address space.

A typical use for multiple threads might be in a file server, in which every incoming request is assigned to a separate thread to work on. That thread might begin processing the request, then block waiting for the disk, then continue work. By splitting the server up into multiple threads, each thread can be purely sequential, even if it has to block waiting for I/O. Nevertheless, all the threads can have access to a single shared software cache. Threads can synchronize using semaphores and mutexes to prevent two threads from accessing the shared cache simultaneously.

Threads are managed and scheduled by the microkernel, so they are not as lightweight as pure user-space threads would be. Nevertheless, thread switching is still reasonably fast. The primary argument for making the threads known to the kernel rather than being pure user concepts relates to our desire to have communication be synchronous (i.e., blocking). In a system in which the kernel knows nothing about threads, when a thread makes what is logically a blocking system call, the kernel must nevertheless return control immediately to the caller, to give the user-space threads package the opportunity to suspend the calling thread and schedule a different thread. Thus a system call that is logically blocking must in fact return control to the "blocked" caller to let the threads package reschedule the CPU. Having the kernel be aware of threads eliminates the need for such awkward mechanisms.

---

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

The second task of the microkernel is to provide low-level memory management. Threads can allocate and deallocate blocks of memory, called *segments*. These segments can be read and written, and can be mapped into and out of the address space of the process to which the calling thread belongs. To provide maximum communication performance, all segments are memory resident.

The third job of the microkernel is to provide the ability for one thread to communicate transparently with another thread, regardless of the nature or location of the two threads. The model used here is a *remote procedure call* (RPC) between a client and a server (Birrell and Nelson, 1984). Conceptually, the initiating thread, called the *client*, calls a library procedure that runs on the *server*. This mechanism is implemented as follows. The client, in fact, calls a local library procedure known as the *client stub* that collects the procedure parameters, builds a header and a buffer, and executes a kernel primitive to perform an RPC. At this point, the calling thread is blocked. The kernel then sends the header and buffer over the network to the destination machine, where it is received by the kernel there. The kernel then passes the header and buffer to a *server stub*, which has previously announced its willingness to receive messages addressed to it. The server stub then calls the server procedure, which does the work requested of it. The reply message follows the reverse route back to the client. When it arrives, the calling thread is given the reply message and is unblocked. The RPC mechanism is illustrated in Fig. 3.

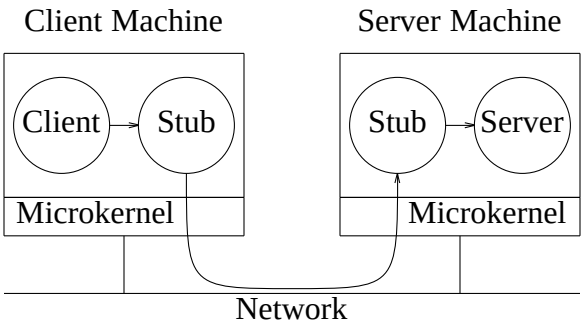

Fig. 3. Remote procedure call.

All RPCs are from one thread to another. User-to-user, user-to-kernel, and kernel-to-kernel communication all occur. (Kernel-to-user is technically legal, but, since that constitutes an upcall, they have been avoided except where that was not feasible). When a thread blocks awaiting the reply, other threads in the same process that are not logically blocked may be scheduled and run.

In order for one thread to send something to another thread, the sender must know the receiver's address. Addressing is done by allowing any thread to choose a random 48-bit number called a *port*. All messages are addressed from a sending port to a destination port. A port is nothing more than a kind of logical thread address. There is no data structure and no storage associated with a port. It is similar to an IP address or an Ethernet address in that respect, except that it is not tied to any particular physical location.

When an RPC is executed, the sending kernel locates the destination port by broadcasting a special LOCATE message, to which the destination kernel responds. Once this cycle has been completed, the sender caches the port, to avoid subsequent broadcasts.

The RPC mechanism makes use of three principal kernel primitives:

- do_remote_op - send a message from client to server and wait for the reply
- get_request - indicates a server's willingness to listen on a port
- put_reply - done by a server when it has a reply to send

Using these primitives it is possible for a server to indicate which port it is listening to, and for clients and servers to communicate. The difference between *do_remote_op* and an RPC is that the former is just the message exchange in both directions, whereas the RPC also includes the parameter packing and unpacking in the stub procedures.

All communication in Amoeba is based on RPC. If the RPC is slow, everything built on it will be slow too (e.g., the file server performance). For this reason, considerable effort has been spent to optimize the performance of the RPC between a client and server running as user processes on different machines, as this is the normal case in a distributed system. In Fig. 4 we give our measured results for sending a zero-length message from a user-level client on one machine to a user-level server on a second machine, plus the sending and receiving of a zero-length reply from the server to the client. Thus it takes 1.1 msec from the time the client thread initiates the RPC until the time the reply arrives and the caller is unblocked. We have also measured the effective data rate from client to server, however, this time using large messages rather than zero-length ones. From the published literature, we have looked for the analogous figures for several other systems and included them for comparison purposes.

System	*Hardware*	*Null RPC in msec.*	*Throughput in kbytes/s*	*Estimated CPU MIPS*	*Implementation Notes*
Amoeba	Sun 3/60	1.1	820	3.0	Measured user-to-user
Cedar	Dorado	1.1	250	4.0	Custom microcode
x-Kernel	Sun 3/75	1.7	860	2.0	Measured kernel-to-kernel
V	Sun 3/75	2.5	546	2.0	Measured user-to-user
Topaz	Firefly	2.7	587	5.0	Consists of 5 VAX CPUs
Sprite	Sun 3/75	2.8	720	2.0	Measured kernel-to-kernel
Mach	Sun 3/60	11.0	?	3.0	Throughput not reported

Fig. 4. Comparative Performance of RPC on Amoeba and other systems.

The RPC numbers for the other systems are taken from the following publications: Cedar (Birrell and Nelson, 1984), *x*-Kernel (Peterson et al, 1990), Sprite (Ousterhout et al., 1988), V (Cheriton, 1988), Topaz (Schroeder and Burrows, 1989), and Mach

(Peterson et al, 1990).

The numbers shown here cannot be compared without knowing about the systems from which they were taken, as the speed of the hardware on which the tests were made varies by about a factor of 3. On all distributed systems of this type running on fast LANs, the protocols are largely CPU bound. Running the system on a faster CPU (but the same network) definitely improves performance, although not linearly with CPU MIPS (Millions of Instructions Per Second) because at some point the network saturates (although none of the systems quoted here even come close to saturating it). As an example, in an earlier paper (Van Renesse et al., 1988), we reported a null RPC time of 1.4 msec, but this was for Sun 3/50s. The current figure of 1.1 msec is for the faster Sun 3/60s.

In Fig. 4 we have not corrected for machine speed, but we have at least made a rough estimate of the raw total computing power of each system, given in the fifth column of the table in MIPS. While we realize that this is only a crude measure at best, we see no other way to compensate for the fact that a system running on a 4 MIPS machine (Dorado) or on a 5 CPU multiprocessor (Firefly) has a significant advantage over slower workstations. As an aside, the Sun 3/60 is indeed faster than the Sun 3/75; this is not a misprint.

Cedar's RPC is about the same as Amoeba's although it was implemented on hardware that is 33 percent faster. Its throughput is only 30% of Amoeba's, but this is partly due to the fact that it used an early version of the Ethernet running at 3 megabits/sec. Still, it does not even manage to use the full 3 megabits/sec.

The *x*-Kernel has a 10% better throughput than Amoeba, but the published measurements are kernel-to-kernel, whereas Amoeba was measured from user process to user process. If the extra overhead of context switches from kernel to user and copying from kernel buffers to user buffers are considered, to make them comparable to the Amoeba numbers, the *x*-kernel performance figures would be reduced to 2.3 msec for the null RPC with a throughput of 748 kbytes/sec when mapping incoming data from kernel to user and 575 kbytes/sec when copying it (L. Peterson, private communication).

Similarly, the published Sprite figures are also kernel-to-kernel. Sprite does not support RPC at the user level, but a close equivalent is the time to send a null message from one user process to another and get a reply, which is 4.3 msec. The user-to-user bandwidth is 170 kbytes/sec (Welch and Ousterhout, 1988).

V uses a clever technique to improve the performance for short RPCs: the entire message is put in the CPU registers by the user process and taken out by the kernel for transmission. Since the 68020 processor has eight 4-byte data registers, up to 32 bytes can be transferred this way. Following the V example, Amoeba does this too.

Topaz RPC was measured on Fireflies, which are VAX-based multiprocessors. The performance shown in Fig. 4 can only be obtained using several CPUs at each end. When only a single CPU is used at each end, the null RPC time increases to 4.8 msec and the throughput drops to 313 kbytes/sec.

The null RPC time for Mach was obtained from a paper published in May 1990 (Peterson et al, 1990) and applies to Mach 2.5, in which the networking code is in the kernel. The Mach RPC performance is worse than any of the other systems by more than a factor of 3 and is ten times slower than Amoeba. A more recent measurement on

an improved version of Mach gives an RPC time of 9.6 msec and a throughput of 250K bytes/sec (R. Draves, private communication).

4. THE AMOEBA SERVERS

Most of the traditional operating system services (such as the directory server) in Amoeba are implemented as server processes. Although it would have been possible to put together a random collection of servers, each with its own model of the world, it was decided early on to provide a single model of what a server does to achieve uniformity and simplicity. That model, and some examples of key Amoeba servers, are described in this section.

4.1. Objects and Capabilities

The basic unifying concept underlying all the Amoeba servers and the services they provide is the *object*. An object is an encapsulated piece of data upon which certain well-defined operations may be performed. It is in essence, an abstract data type. Objects are passive. They do not contain processes or methods or other active entities that "do" things. Instead, each object is managed by a server process. To perform an operation on an object, a client does an RPC with the server, specifying the object, the operation to be performed, and optionally, any parameters needed.

Objects are named and protected by special tickets called *capabilities*. To create an object, a client does an RPC with the appropriate server specifying what it wants. The server then creates the object and returns a 128-bit capability to the client. On subsequent operations, the client must present the capability to identify the object. The format of a capability is shown in Fig. 5

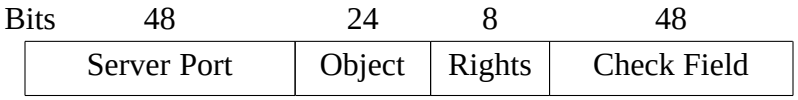

Fig. 5. A capability.

When a client wants to perform an operation on an object, it calls a stub procedure that builds a message containing the object's capability, and then traps to the kernel. The kernel extracts the *Server Port* field from the capability and looks it up in its cache to locate the machine on which the server resides. If there is no cache entry, or that entry is no longer valid, the kernel locates the server by broadcasting.

The rest of the information in the capability is ignored by the kernels and passed to the server for its own use. The *Object* field is used by the server to identify the specific object in question. For example, a file server might manage thousands of files, with the object number being used to tell it which one is being operated on. In a sense, the *Object* field is analogous to the *i-node number* in UNIX.

The *Rights* field is a bit map telling which of the allowed operations the holder of a capability may perform. For example, although a particular object may support reading and writing, a specific capability may be constructed with all the rights bits except

READ turned off.

The *Check Field* is used for validating the capability. Capabilities are manipulated directly by user processes. Without some form of protection, there would be no way to prevent user processes from forging capabilities. The basic algorithm is as follows. When an object is created, the server picks a random *Check Field* and stores it both in the new capability and inside its own tables. All the rights bits in a new capability are initially on, and it is this *owner capability* that is returned to the client. When the capability is sent back to the server in a request to perform an operation, the *Check Field* is verified.

To create a restricted capability, a client can pass a capability back to the server, along with a bit mask for the new rights. The server takes the original *Check Field* from its tables, EXCLUSIVE ORs it with the new rights (which must be a subset of the rights in the capability), and then runs the result through a one-way function. Such a function, $y = f(x)$, has the property that given x it is easy to find y, but given only y, finding x requires an exhaustive search of all possible x values (Evans et al., 1974).

The server then creates a new capability, with the same value in the *Object* field, but the new rights bits in the *Rights* field and the output of the one-way function in the *Check Field*. The client may give this to another process, if it wishes. When the capability comes back to the server, the server sees from the *Rights* field that it is not an *owner capability* because at least one bit is turned off. The server then fetches the original random number from its tables, EXCLUSIVE ORs it with the *Rights* field, and runs the result through the one-way function. If the result agrees with the *Check Field*, the capability is accepted as valid. It should be obvious from this algorithm that a user who tries to add rights that he does not have will simply invalidate the capability. Capabilities are used throughout Amoeba for both naming of all objects and for protecting them. This single mechanism leads to an easy-to-understand and easy-to-implement naming and protection scheme. It also is fully location transparent. To perform an operation on an object, it is not necessary to know where the object resides. In fact, even if this knowledge were available, there would be no way to use it.

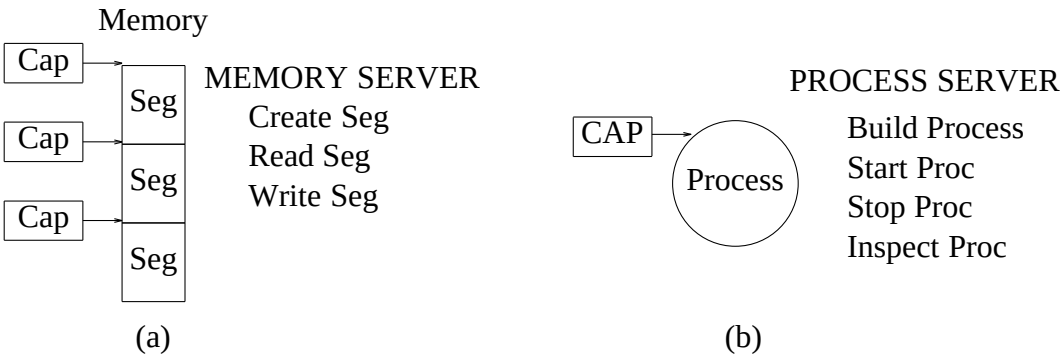

Fig. 6. Typical capability usage.

Note that Amoeba does not use access control lists for authentication. The protection scheme used requires almost no administrative overhead. However, in an insecure

environment, cryptography may be required to keep capabilities from being accidentally disclosed.

In Fig. 6, we show two examples of how capabilities are used in Amoeba. In (a), a group of three segments have been created, each of which has its own capability. Using these capabilities, the process creating the segments can read and write the segments. Given a collection of memory segments, a process can go to the process server and ask for a process to be constructed from them, as shown in (b). This results in a process capability, through which the new process can be run, stopped, inspected, and so on. This mechanism for process creation is much more location transparent and efficient in a distributed system than the UNIX *fork* system call.

4.2. The Bullet Server

Like all operating systems, Amoeba has a file system. However, unlike most other ones, the choice of file system is not dictated by the operating system. The file system runs as a collection of server processes. Any user who does not like the standard ones is free to write his own. The microkernel does not know, or care, which one is the "real" file system. In fact, different users may use different and incompatible file systems at the same time, if they so desire. In this section we will describe an experimental file server called the *bullet server*, which has a number of interesting properties.

The bullet server was designed to be very fast (hence the name). It was also designed to run on future machines having large primary memory, rather than low-end machines where memory is very tight. The organization is quite different from most conventional file servers. In particular, files are *immutable*. Once a file has been created, it cannot subsequently be changed. It can be deleted, and a new file created in its place, but the new file has a different capability thanthe old one. This fact simplifies automatic replication, as will be seen. In effect, there are only two major operations on files: CREATE and READ.

Because files cannot be modified after their creation, the size of a file is always known at creation time. This allows files to be stored contiguously on the disk, and also in the in-core cache. By storing files contiguously, they can be read into memory in a single disk operation, and they can be sent to users in a single RPC reply message. These simplifications lead to high performance.

The bullet server maintains a table with one entry per file, analogous to the UNIX i-node table. When a client process wants to read a file, it sends the capability for the file to the bullet server. The server extracts the object number from the capability and uses it as an index into the in-core i-node table to locate the entry for the file. The entry contains the random number used in the *Check Field* as well as some accounting information and two pointers: one giving the disk address of the file and one giving the cache address (if the file is in the cache). This design, shown in Fig. 7, leads in principle to a simple implementation and high performance. It is well suited to optical juke boxes and other write-once media, and can be used as a base for more sophisticated storage systems.

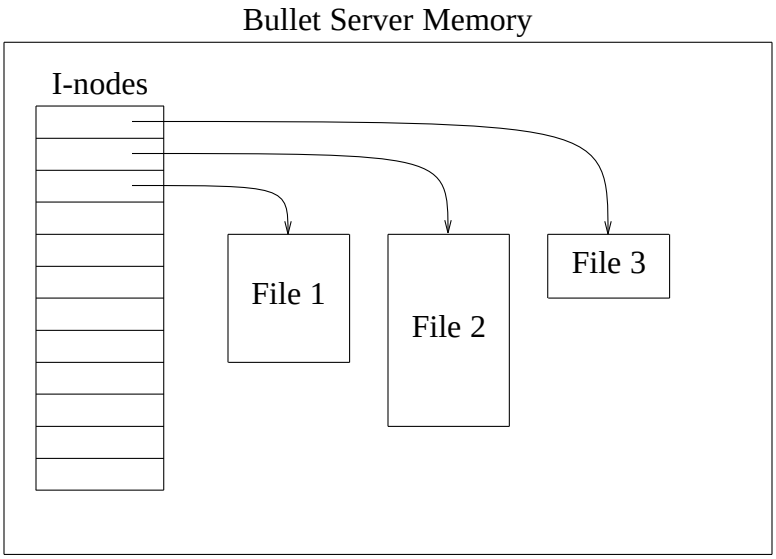

Fig. 7. The bullet server.

4.3. The Directory Server

Another interesting server is the *directory server*. Its primary function is to provide a mapping from human-readable (ASCII) names to capabilities. Users can create one or more directories, each of which contains multiple (name, capability-set) pairs. Operations are provided to create and delete directories, add and delete (name, capability-set) pairs, and look up names in directories. Unlike bullet files, directories are not immutable. Entries can be added to existing directories and entries can be deleted from existing directories.

The layout of an example directory with six entries is shown in Fig. 8. This directory has one row for each of the six file names stored in it. The directory also has three columns, each one representing a different protection domain. For example, the first column might store capabilities for the owner (with all the rights bits on), the second might store capabilities for members of the owner's group (with some of the rights bits turned off), and the third might store capabilities for everyone else (with only the read bit turned on). When the owner of a directory gives away a capability for it, the capability is really a capability for a single column, not for the directory as a whole. When giving a directory capability to an unrelated person, the owner could give a capability for the third column, which contains only the highly restricted capabilities. The recipient of this capability would have no access to the more powerful capabilities in the first two columns. In this manner, it is possible to approximate the UNIX protection system, as well as devise other ones for specific needs.

Another important aspect of the directory server is that the entry for a given name in a given column may contain more than one capability. In principle it is a capability set, that is, a group of capabilities for replicas of the file. Because files are immutable, when a file is created, it is possible to install the newly generated capability in a

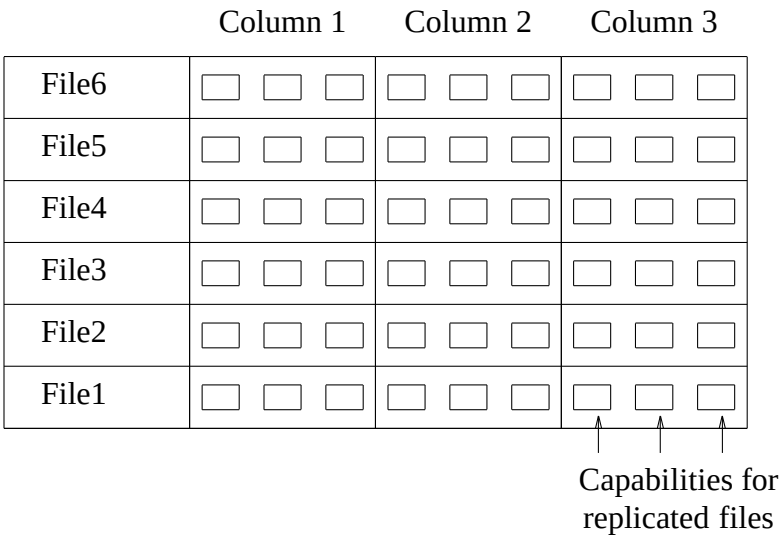

Fig. 8. The directory server.

directory, with the understanding that in due time, a specific number of replicas will be automatically generated and added to the entry. Thus an entry in a directory consists of a set of capabilities, all for the same file, and normally located on different bullet or other servers.

When a user presents the directory server with a capability for a (column of a) directory, along with an ASCII name, the server returns the capability set corresponding to that name and column. The user can then try any one of the servers to access the file. If that one is down, it can try one of the others. In this way, an extremely high availability can be achieved. The capability-set mechanism can be made transparent for users by hiding it in library procedures. For example, when a file is opened, the open procedure could fetch and store the capability set internally. Subsequently, the read procedure could keep trying capabilities until it found a functioning server. The key to the whole idea is that files are immutable, so that the replication mechanism is not subject to race conditions and it does not matter which capability is used, since the files cannot change.

4.4. The Boot Server

As a final example of an Amoeba server, let us consider the *boot server*. The boot server is used to provide a degree of fault tolerance to Amoeba by checking that all servers that are supposed to be running are in fact running, and taking corrective action when they are not. A process that is interested in surviving crashes can register with the boot server. They agree on how often the boot server should poll, what it should send, and what reply it should get. As long as the server responds correctly, the boot server takes no further action.

However, if the server should fail to respond after a specified number of attempts, the boot server declares it dead, and arranges to allocate a new pool processor on which a new copy is started. In this manner, critical services are automatically rebooted if they

should ever fail. The boot server could itself be replicated, to guard against its own failure (although this is not done at present).

5. APPLICATIONS OF AMOEBA

Although Amoeba has other servers, let us now briefly turn to some applications of Amoeba. One application is to use it as a program development environment. A second is to use it for parallel programming. A third is to use it in embedded industrial applications. In the following sections we will discuss each of these in turn.

5.1. Amoeba as a Program Development Environment

To make Amoeba suitable for program development, we have written a partial UNIX emulation library and written or ported numerous UNIX application programs. We have not aimed at binary compatibility with UNIX, since our primary goal was to do research in the next generation of operating systems. In this respect, having to be binary compatible with UNIX would have meant taking the bad along with the good. It is difficult to do innovative research with such restrictions.

Instead, we have opted for writing a set of library procedures that emulate most of the common UNIX system calls, such as OPEN, READ, WRITE, CLOSE, FORK, and so on. The ultimate goal is POSIX P1003.1 conformance, although that has not yet been achieved. Each library procedure performs its work by making calls on the Amoeba servers. How it does this varies from procedure to procedure. For example, the usual way the file system calls are handled is to read the file into the caller's address space in its entirety (if possible), operate on it locally, and then write it back to the bullet server in a single CREATE operation. Finally, the new capability is installed in the proper directory, removing the old one. Then the old file can be deleted by the bullet server. Only if the file is too large for memory is a different procedure followed. This scheme is not unlike the Andrew file system (Howard et al, 1988).

Since the UNIX file system is being supported on top of the bullet server, the latter's file replication facility is automatically present, without any special effort.

Amoeba and its servers are largely stateless, whereas various aspects of UNIX require maintaining state information. This gap has been bridged by having a *session server* that keeps track of the state for the current UNIX login session.

Many UNIX-like utilities are available with Amoeba (well over 100). Some of these have been taken from MINIX (Tanenbaum, 1987). Others are public domain. Still others have been written from scratch. All of the compilers (C, Pascal, Modula 2, Orca) have been written by us using our own compiler writing system (Tanenbaum et al., 1983). Thus none of the utilities, libraries, compilers, operating system or any other Amoeba programs contain any UNIX code whatsoever. As a result, an AT&T license is not required for Amoeba. Amoeba is distributed with full source code.

The above notwithstanding, Amoeba is not really an attempt to simply redo UNIX. It is designed as a research vehicle in distributed operating systems, languages and applications. Some of these are UNIX-like; others are completely new. As an example of an extension to a standard UNIX application, we have written a new, parallel version of *make*, called *amake* (Baalbergen et al., 1989). When multiple compilations must be run to produce a given *a.out* file, *amake* runs them in parallel, to gain speed, as shown in

Fig. 9. Another interesting property of *amake*, is that it does not use a traditional *Makefile*, but collects the dependency information on its own and maintains it in a hidden directory. This feature makes it easier to use than conventional *make*, which, as an aside, is also present.

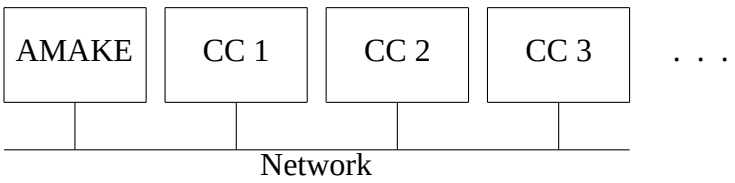

Fig. 9. *Amake* allows multiple compilations to run in parallel.

5.2. Parallel Programming

Another use of Amoeba is for supporting parallel programming. As can be seen in Fig. 1, the processor pool model is an attractive way to exploit massive parallelism. A chess program, for example, has been written to allow each of the pool processors to evaluate different parts of the game tree in parallel. The various processors communicate using RPC and other techniques.

To ease the work of producing parallel programs, we have designed and implemented a new language called *Orca* (Bal, 1990; Bal and Tanenbaum, 1991). The idea underlying Orca is that programmers should be able to define shared data objects upon which specific operations are defined (in effect, abstract data types). Process on different processors can share these abstract objects, even though the system itself does not necessarily contain any physical shared memory. How this illusion is supported is the job of Amoeba and the Orca run-time system.

To make this point clearer, let us discuss one possible implementation of the Orca run-time system. This implementation is not present in the initial release of Amoeba (Version 4.0, Spring 1991), but will be present in Version 5.0. The basic idea is that each shared object is replicated in full on all processors that are running a process interested in the shared object (Bal and Tanenbaum, 1991). Orca operations are divided into two categories: those involving only reading the object, and those that change the object.

The read operations are easy. Since a copy of the object is located on each machine, the operation can be performed entirely locally, with no network traffic. This means that they can be performed with no delay, highly efficiently.

Operations that involve changing an object are more complicated. The algorithm used is based on one of the services offered by Amoeba 5.0, *reliable broadcast* (Kaashoek and Tanenbaum, 1991). By this we mean, a message from one sender can be sent to a group of receivers with certainty that all of them will receive it (unless some of them crash). The mechanism for achieving reliable broadcasting will be described below.

Given the existence of reliable broadcasting as a primitive, the way shared objects are updated is straightforward. To update an object, the new value is just broadcast to all sites holding the object. Alternatively, the operation code and its parameters can be broadcast, and each site can carry out the operation locally.

Reliable broadcasting is implemented as follows. One node is chosen as the *sequencer*. If this node should ever fail, a new sequencer is chosen. Although we do not have space to describe it here, the protocol has been designed to withstand an arbitrary number of failures. To do a reliable broadcast, a process can send a point-to-point message to the sequencer, which then adds a sequence number to the header and broadcasts it. When a processor receives a broadcast message, it checks the sequence number. If it is the expected one, the message is accepted. If one or more messages have been missed, the processor asks the sequencer to send it the missing messages. In all cases, messages are only passed to the application in strict sequence, with no gaps. More details are given in (Kaashoek et al, 1989).

Now let us briefly consider a typical parallel application that runs on Amoeba: the traveling salesman problem (TSP). The goal is for a salesman to start out in a certain city, and then visit a specific list of other cities exactly one time each, covering the shortest possible path in doing so. To illustrate how this problem can be solved in parallel, consider a salesman starting in Amsterdam, and visiting New York, Tokyo, Sydney, Nairobi, and Rio de Janeiro. Processor 1 could work on all paths starting Amsterdam-New York. Processor 2 could work on all paths starting Amsterdam-Tokyo. Processor 3 could work on all paths starting Amsterdam-Sydney, and so on.

Although letting each processor work independently on its portion of the search tree is feasible and will produce the correct answer, a far more efficient technique is to use the well-known branch and bound method (Lawler and Wood, 1966). To use this method, first a complete path is found, for example by using the shortest-flight-next algorithm. Whenever a partial path is found whose length exceeds that of the currently best known total path, that part of the search tree is truncated, as it cannot produce a solution better than what is already known. Similarly, whenever a new path is found that is shorter than the best currently known path, this new path becomes the current best known path.

The Orca approach to solving the traveling salesman problem using branch and bound is to have a shared object that contains the best known path and its length. As usual with Orca objects, this object is replicated on all processors working on the problem. Two operations are defined on this object: reading it and updating it. As each processor examines its part of the search tree, it keeps checking (the local value of) the best known path, to see if the path it is investigating is still a possible candidate. This operation, which occurs very frequently, is a local operation, not requiring any network traffic.

When a new best path is found, the processor that found it performs the update operation, which triggers a reliable broadcast to update all copies of the best path on all processors. Update operations are always serializable (in the data base sense). If two processes perform simultaneous updates on the same object, one of them goes first and completes its work before the other is allowed to begin. When the second one begins, it sees the value the first one stored. In this way, race conditions are avoided. The measured performance of the TSP implementation is shown in Fig. 10.

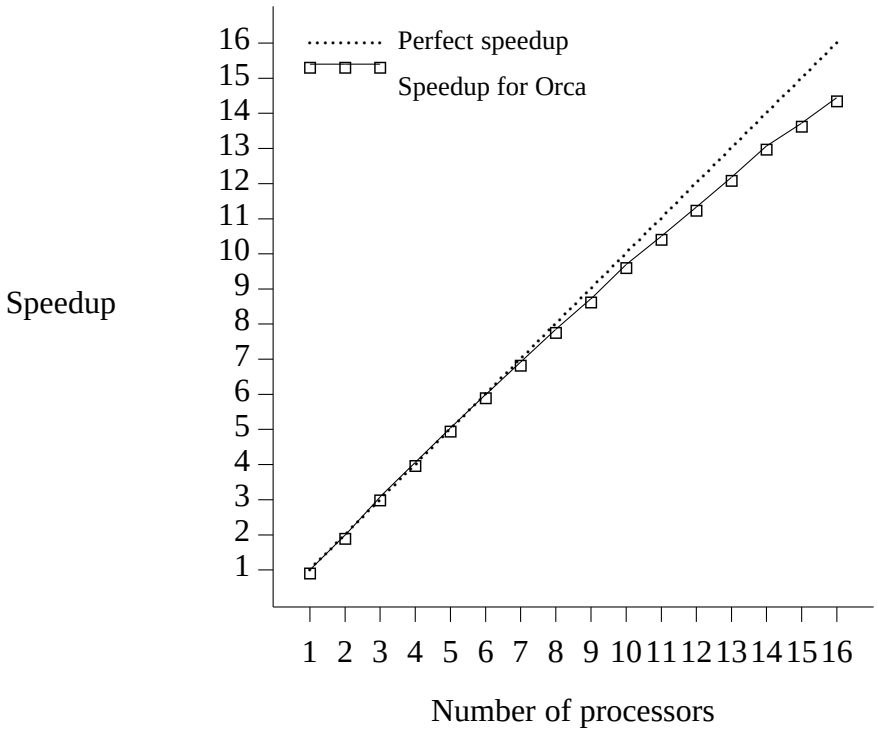

Fig. 10. Speedup of TSP in Orca

5.3. Amoeba in an Industrial Environment

As another example of how Amoeba is currently being used, let us consider an example from the European space industry. In this application, a substantial number of television cameras are to be carried aloft in a spacecraft. Each camera will observe one or more experiments, so that the scientific investigators on the ground will be able to monitor and interact with their experiments in space. Each camera will be controlled by a computer. These computers will contain special boards that perform analog to digital conversion of the incoming television signal. Once the signal has been transformed to digital form, the bits will be moved over a local area network in software, and eventually transmitted to the ground.

A testbed for this system has already been constructed and is in operation, as illustrated in Fig. 11. Amoeba was chosen as the distributed operating system due to its high performance. From Fig. 4 it can be seen that the throughput between two Amoeba user processes is over 6 megabits/sec (on a 10 megabit/sec Ethernet). In Amoeba 5.0, this figure is over 8 megabits/sec continuous throughput, which is over 80% of the theoretical capacity of the Ethernet, a figure achieved by few other systems.

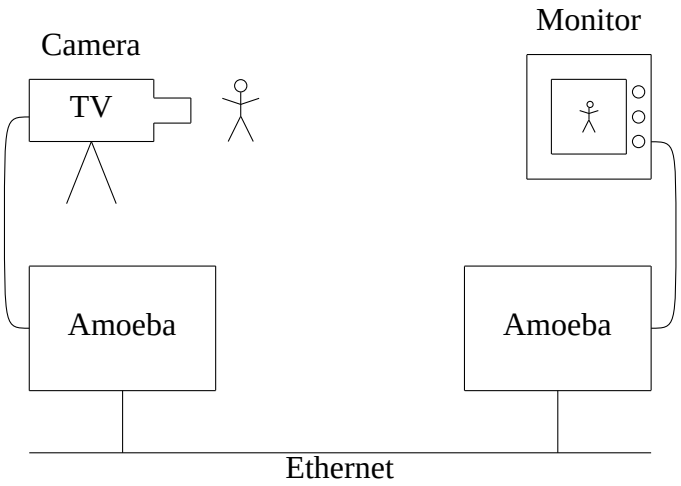

Fig. 11. Amoeba in the space testbed.

6. AMOEBA ON A WIDE-AREA NETWORK

Although Amoeba has been primarily used to date on local area networks, there has also been some work with it on wide-area networks. In this section we will discuss how it is done. As far as Amoeba is concerned, the main difference between a LAN and a WAN is the lack of broadcasting on a WAN. When a process performs an RPC using a capability whose port has not previously been used, the kernel on that machine locates the destination by broadcasting a special LOCATE packet. On a wide area network, such broadcasts are not possible, so a slightly different approach is taken, one that nevertheless preserves the goal of transparency—the client cannot tell where the server is, even if it is located in a different country, and all actions taken by both client and server are the same, whether they are on the same network or not.

The Amoeba approach to wide area networks is to require services that want to be known over a WAN to *publish* their port. Publishing a port is done by the (human) owner of the service, not by the server code. To publish the port, the owner runs a special program that sends the server's port and network address to the set of gateways (see Fig. 1) on whose networks the server is to be known. When such a request is received, a special *server agent* process is created on the gateway machine. This server agent listens to the server's port. When a client on the server agent's LAN does an RPC to the server, the client's kernel broadcasts a LOCATE packet, which is received by the gateway's kernel. The gateway's kernel responds in the usual way, and the RPC arrives at the server agent.

The server agent then passes it to a *link* process, which transmits it over the wide-area link using whatever protocol is required there. At the destination, a *client agent* is created, which does an RPC with the server. The reply follows the reverse path back to the client. The beauty of this scheme, shown in Fig. 12, is that neither the client nor the server processes know that their RPCs are in any way strange. To them, everything looks local. Only the agents and link processes know that wide area communication is involved. Thus the LAN protocols are in no way affected by the WAN protocol, which

can be changed at will without affecting the local protocol. Only the link processes have to be adapted. Our most heavily used link process at present is for TCP/IP.

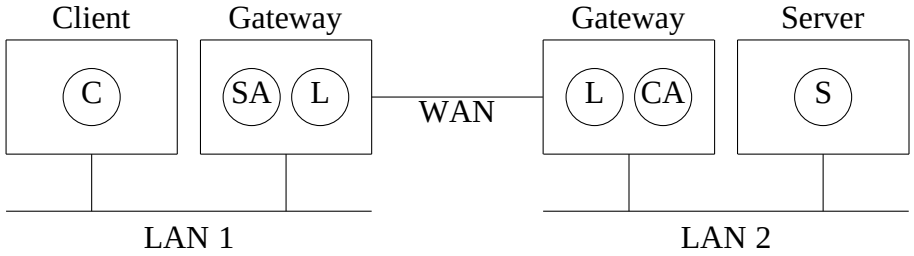

Fig. 12. Amoeba on a wide-area network. C = Client, SA = Server Agent, L = Link, CA = Client Agent, and S = Server.

7. DISCUSSION

In this section we will discuss some of the lessons we have learned with Amoeba and some of the changes we are making in the next version (5.0) based on these lessons. One area where little improvement is needed is portability. Amoeba started out on the 680x0 CPUs, and has been moved to the VAX, NS 32016, Intel 80386, SPARC and MIPS processors.

The use of a microkernel has been very satisfactory. A microkernel-based design is simple and flexible. The potential fear that it would be too slow for practical use has not been borne out. By making it possible to have servers that run as user processes, we have been able to easily experiment with different kinds of servers and to write and debug them without having to bring down and reboot the kernel all the time.

For the most part, RPC communication has been satisfactory, as have the three primitives for accessing it. Occasionally, however, RPC gives problems in situations in which there is no clear master-slave relation, such as in UNIX pipelines (Tanenbaum and van Renesse, 1988). Another difficulty is the fact that although RPC is fine for sending a message from one sender to one receiver, it is less suited for group communication. This limitation will be eliminated in Amoeba 5.0 with the introduction of reliable broadcasting as a fundamental primitive.

The object-based model for services has also worked well. It has given us a clear model to work with when designing new servers. The use of capabilities for transparent naming and protection can also be largely regarded as a success. It is conceivable, however, that if the system were to be extended to millions of machines worldwide, the idea of using capabilities would have to be revisited. The fear is that casual users might be too lax about protecting their capabilities. On the other hand, they might come to regard them like the codes they use to access automatic bank teller machines, and take good care of them. We do not really know.

In Amoeba 4.0, when a thread is started up, it runs until it logically blocks or exits. There is no pre-emption. This was a serious error. The idea behind it was that once a thread starting using some critical table, it would not be interrupted by another thread in the same process until it finished. This scheme seemed simple to understand, and it was

certainly easy to program. Problems arose when programmers put print statements in critical regions for debugging, not realizing that the print statements did RPCs with remote terminal servers, thus allowing the thread to be rescheduled and thus breaking the sanctity of the critical region. In Amoeba 5.0, all threads will be preemptable, and programmers will be required to protect critical regions with mutexes and semaphores.

One area of the system which we think has been quite innovative is the design of the file server and directory server. We have separated out two distinct parts, the bullet server, which just handles storage, and the directory server, which handles naming and protection. The bullet server design allows it to be extremely fast, while the directory server design gives a flexible protection scheme and also supports file replication in a simple and easy to understand way. The key element here is the fact that files are immutable, so they can be replicated at will, and copies regenerated if necessary.

On the other hand, for applications such as data bases, having immutable files is clearly a nuisance as they cannot be updated. A separate data base server (that does not use the bullet server) is certainly one possibility, but we have not investigated this in detail. Append-only log files are also difficult to handle with the bullet server.

The Amoeba 4.0 UNIX emulation was done with the idea of getting most of the UNIX software to work without too much effort on our part. The price we pay for this approach is that we will probably never be able to provide 100% compatibility. For example, the whole concept of uids and gids is very hard to get right in a capability-based system. Our view of protection is totally different. Still, since our goal was to do research on new operating systems rather than provide a plug-to-plug UNIX replacement, we consider this acceptable.

Although Amoeba was originally conceived as a system for *distributed* computing, the existence of the processor pool with many CPUs close together has made it quite suitable for *parallel* computing as well. That is, we have become much more interested in using the processor pool to achieve large speedups on a single problem. The use of Orca and its globally shared objects has been a big help. All the details of the physical distribution are completely hidden from the programmer. Initial results indicate that almost linear speedup can be achieved on some problems involving branch and bound, successive overrelaxation, and graph algorithms.

Performance, has been good in various ways. The minimum RPC time for Amoeba is 1.1 msec between two user-space processes on Sun 3/60s, and interprocess throughput is nearly 800 kbytes/sec. The file system lets us read and write files at about the same rate (assuming cache hits on the bullet server). On the other hand, the UNIX FORK system call is slow, because it requires making a copy of the process remotely, only to have it exit within a few milliseconds.

Amoeba originally had a homebrew window system. It was faster than X-windows, and in our view, cleaner. It was also much smaller and much easier to understand. For these reasons we thought it would be easy to get people to accept it. We were wrong. We have since switched to X windows.

8. COMPARISON WITH OTHER SYSTEMS

In some ways, Amoeba resembles other well-known distributed systems, such as Mach (Accetta et al, 1986), Chorus (Rozier, 1988) V (Cheriton, 1988) and Sprite (Ousterhout et al., 1988). Although a comprehensive comparison of Amoeba with these would no doubt be very interesting, space limitations prohibit us from doing that here. In (Douglis et al., 1990), we have provided a detailed comparison between Amoeba and Sprite, however. Nevertheless, we would like to make a few general remarks.

The goals of the Amoeba project differ somewhat from the goals of the other systems. We were interested in doing research on distributed systems, and building a good testbed for experimenting with distributed systems, algorithms, languages and applications. Although we are fully aware how popular UNIX is in some circles, it was never our intention to have Amoeba be a plug compatible replacement for UNIX as has been the case with some other systems. Nevertheless, providing more UNIX compatibility is certainly a possibility for the future.

Another difference with other systems is our emphasis on Amoeba as a *distributed* system. It was intended from the start to run Amoeba on a large number of machines. One comparison with Mach is instructive on this point. Mach uses a clever optimization to pass messages between processes running on the same machine. The page containing the message is mapped from the sender's address space to the receiver's address space, thus avoiding copying. Amoeba does not do this because we consider the key issue in a distributed system the communication speed between processes running on different machines. That is the normal case. Only rarely will two processes happen to be on the same physical processor in a true distributed system, especially if there are hundreds or thousands of processors, so we have put a lot of effort into optimizing the distributed case, not the local case. This is clearly a philosophical difference.

9. CONCLUSION

The Amoeba project has clearly demonstrated that it is possible to build an efficient, high-performance distributed operating system. By having a microkernel, most of the key features are implemented as user processes, which means that the system can evolve gradually as needs change and we learn more about distributed computing. The object-based nature of the system, and the use of capabilities provide a unifying theme that holds the various pieces together.

Amoeba has gone through four major versions in the past years. Its design is clean and its performance in many areas is good. By and large we are satisfied with the results. Nevertheless, no operating system is ever finished, so we are continually working to improve it. Amoeba is now available. For information on how to obtain it, please contact the first author, preferably by electronic mail at ast@cs.vu.nl.

10. ACKNOWLEDGEMENTS

We would like to thank Sape Mullender, Guido van Rossum, Jack Jansen, and the other people at CWI who made significant contributions to the development of Amoeba. In addition, Leendert van Doorn provided valuable feedback about the paper.

11. REFERENCES

Accetta, M., Baron, R., Bolosky W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A New Kernel Foundation for UNIX Development. *Proceedings of the Summer Usenix Conference*, Atlanta, GA, (July 1986)

Baalbergen, E.H, Verstoep, K., and Tanenbaum, A.S. On the Design of the Amoeba Configuration Manager. *Proc. 2nd Int'l Workshop on Software Config. Mgmt.*, ACM, (1989).

Bal, H.E.: *Programming Distributed Systems* Summit NJ: Silicon Press, (1990).

Bal, H.E., and Tanenbaum, A.S. Distributed Programming with Shared Data, *Computer Languages*, vol. 16, pp. 129-146, Feb. 1991.

Birrell, A.D., and Nelson, B.J. Implementing Remote Procedure Calls, *ACM Trans. Comput. Systems* 2, (Feb. 1984) pp. 39-59.

Cheriton, D.R. The V Distributed System. *Comm. ACM* 31, (March 1988), pp. 314-333.

Douglis, F., Kaashoek, M.F., Tanenbaum, A.S., and Ousterhout, J.K.: A Comparison of Two Distributed Systems: Amoeba and Sprite. Report IR-230, Dept. of Mathematics and Computer Science, Vrije Universiteit, (Dec. 1990).

Evans, A., Kantrowitz, W., and Weiss, E. A User Authentication Scheme Not Requiring Secrecy in the Computer. *Commun. ACM* 17, (Aug. 1974), pp. 437-442.

Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., and Sidebotham, R.N.: Scale and Performance in a Distributed File System. *ACM Trans. on Comp. Syst.* 6, (Feb. 1988), pp. 55-81.

Kaashoek, M.F., and Tanenbaum, A.S.: "Group Communication in the Amoeba Distributed Operating System" *Proc. 11th Int'l Conf. on Distr. Comp. Syst.* IEEE, (May 1991).

Kaashoek, M.F., Tanenbaum, A.S., Flynn Hummel, S., and Bal, H.E. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, vol. 23, (Oct 1989), pp. 5-19.

Lawler, E.L., and Wood, D.E. Branch and Bound Methods A Survey. *Operations Research* 14, (July 1966), pp. 699-719.

Mullender, S.J., van Rossum, G., Tanenbaum, A.S., van Renesse, R., van Staveren, J.M. Amoeba — A Distributed Operating System for the 1990s. *IEEE Computer* 23, (May 1990), pp. 44-53.

Ousterhout, J.K., Cherenson, A.R., Douglis, F., Nelson, M.N., and Welch, B.B. The Sprite Network Operating System. *IEEE Computer* 21, (Feb. 1988), pp. 23-26.

Peterson, L., Hutchinson, N., O'Malley, S., and Rao, H. The x-kernel: A Platform for Accessing Internet Resources. *IEEE Computer* 23 (May 1990), pp. 23-33.

Rozier. M, Abrossimov. V, Armand. F, Boule. I, Gien. M, Guillemont. M, Hermann. F, Kaiser. C, Langlois. S, Leonard, P., and Neuhauser. W. CHORUS Distributed Operating System. *Computing Systems* 1 (Fall 1988), pp. 299-328.

Schroeder, M.D., and, Burrows, M. Performance of the Firefly RPC. *Proc. Twelfth ACM Symp. on Oper. Syst. Prin.*, ACM, (Dec. 1989), pp. 83-90.

Tanenbaum, A.S. A UNIX Clone with Source Code for Operating Systems Courses. *Operating Syst. Rev.* 21, (Jan. 1987), pp. 20-29.

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

Tanenbaum, A.S., Renesse, R. van, Staveren, H. van., Sharp, G.J., Mullender, S.J., Jansen, J., and Rossum, G. van: "Experiences with the Amoeba Distributed Operating System," *Commun. of the ACM* vol. 33, (Dec. 1990), pp. 46-63.

Tanenbaum, A.S., van Staveren, H., Keizer, E.G., and Stevenson, J.W.: "A Practical Toolkit for Making Portable Compilers," *Commun. ACM*, vol. 26, pp. 654-660, Sept. 1983.

Van Renesse, R., Van Staveren, H., and Tanenbaum, A.S. Performance of the Amoeba Distributed Operating System. *Software—Practice and Experience* 19, (March 1989) pp. 223-234.

Welch, B.B. and Ousterhout, J.K. Pseudo Devices: User-Level Extensions to the Sprite File System. *Proc. Summer USENIX Conf.*, (June 1988), pp. 37-49.