Experiences with the Amoeba Distributed Operating System

Andrew S. Tanenbaum
Robbert van Renesse[1]
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

Sape J. Mullender[2]
Jack Jansen
Guido van Rossum

Centrum voor Wiskunde en Informatica
Kruislaan 413
1098 SJ Amsterdam, The Netherlands
Internet: sape@cwi.nl, jack@cwi.nl, guido@cwi.nl

ABSTRACT

The Amoeba distributed operating system has been in development and use for over eight years now. In this paper we describe the present system and our experience with it—what we did right, but also what we did wrong.

Computing Reviews categories: C.2.4, D.4

Keywords: Operating systems, Distributed systems, Distributed operating systems, Computer networks, Experience

Descriptors: Network operating systems, Distributed applications, Distributed systems, Measurements

General terms: Design, Experimentation, Performance

1. INTRODUCTION

The Amoeba project is a research effort aimed at understanding how to connect multiple computers together in a seamless way (Mullender and Tanenbaum, 1986; Tanenbaum et al., 1986; Tanenbaum and van Renesse, 1985). The basic idea is to provide the users with the

1. This research was supported in part by the Netherlands Organization for Scientific Research (N.W.O.) under grant 125-30-10.

illusion of a single powerful timesharing system, when, in fact, the system is implemented on a collection of machines, potentially distributed among several countries. This research has led to the design and implementation of the Amoeba distributed operating system, which is being used as a prototype and vehicle for further research. In this paper we will describe the current state of the system (Amoeba 3.0), and tell some of the lessons we have learned designing and using it over the past eight years. We will also discuss how this experience has influenced our plans for the next version, Amoeba 4.0.

Amoeba was originally designed and implemented at the Vrije Universiteit in Amsterdam, and is now being jointly developed there and at the Centre for Mathematics and Computer Science, also in Amsterdam. The chief goal of this work is to build a distributed system that is *transparent* to the users. This concept can best be illustrated by contrasting it with a network operating system, in which each machine retains its own identity. With a network operating system, each user logs into one specific machine, his home machine. When a program is started, it executes on the home machine, unless the user gives an explicit command to run it elsewhere. Similarly, files are local unless a remote file system is explicitly mounted or files are explicitly copied. In short, the user is clearly aware that multiple independent computers exist, and must deal with them explicitly.

In a transparent distributed system, in contrast, users effectively log into the system as a whole, and not to any specific machine. When a program is run, the system, not the user, decides the best place to run it. The user is not even aware of this choice. Finally, there is a single, system wide file system. The files in a single directory may be located on different machines possibly in different countries. There is no concept of file transfer, uploading or downloading from servers, or mounting remote file systems. The fact that a file is remote is not visible to the user at all.

The remainder of this paper will describe Amoeba and the lessons we have learned from building it. In Section 2, we will give a technical overview of Amoeba as it currently stands. Since Amoeba uses the client-server model, in Section 3 we will describe some of the more important servers that have been implemented so far. This is followed by a description of how wide-area networks are handled, in Section 4. In Section 5 we will discuss a number of applications that run on Amoeba. Measurements have shown Amoeba to be extremely fast, so in Section 6 we will present some of these measurements. Finally, in Section 7, we will discuss the successes and failures that we have encountered over the past 5 years, so that others may profit from those ideas that have worked out well and avoid those that have not.

2. TECHNICAL OVERVIEW OF AMOEBA

Before describing the software, it is worth saying something about the system architecture on which Amoeba runs.

2. The research at CWI was supported in part by a grant from Digital Equipment Corporation.

2.1. System Architecture

The Amoeba architecture consists of four principal components, as shown in Fig. 1. First are the workstations, one per user, on which users can carry out editing and other tasks that require fast interactive response. The workstations are all diskless, and are primarily used as intelligent terminals that do window management, rather than as computers for running complex user programs. We are currently using SUN-3s and VAXstations as workstations. In the next generation of hardware we may use X-terminals.

Second are the pool processors, a group of CPUs that can be dynamically allocated as needed, used, and then returned to the pool. For example, the *make* command might need to do six compilations, so six processors could be taken out of the pool for the time necessary to do the compilation and then returned. Alternatively, with a five-pass compiler, 5 x 6 = 30 processors could be allocated for the six compilations, gaining even more speedup. Many applications, such as heuristic search in AI applications (e.g., playing chess), use large numbers of pool processors to do their computing. We currently have 48 single board VME-based computers using the 68020 and 68030 CPUs. We also have 10 VAX CPUs forming an additional processor pool.

Third are the specialized servers, such as directory servers, file servers, data base servers, boot servers, and various other servers with specialized functions. Each server is dedicated to performing a specific function. In some cases, there are multiple servers that provide the same function, for example, as part of the replicated file system.

Fourth are the gateways, which are used to link Amoeba systems at different sites and different countries into a single, uniform system. The gateways isolate Amoeba from the peculiarities of the protocols that must be used over the wide-area networks.

All the Amoeba machines run the same kernel, which primarily provides multithreaded processes, communication services, and little else. The basic idea behind the kernel was to keep it small, to enhance its reliability, and to allow as much as possible of the operating system to run as user processes, providing for flexibility and experimentation.

2.2. Objects and Capabilities

Amoeba is an object-based system. 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* (Dennis and Van Horn, 1966), 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.

B: box wid 1.4i ht 2.2i "Processor Pool" at last box.n above L1: line right 1i with .start at last box.nw + (0.2i, -0.5i) line up 0.3i with .start at 1/11 <L1.start, L1.end> line up 0.3i with .start at 2/11 <L1.start, L1.end> line up 0.3i with .start at 3/11 <L1.start, L1.end> line up 0.3i with .start at 4/11 <L1.start, L1.end> line up 0.3i with .start at 5/11 <L1.start, L1.end> line up 0.3i with with .start at 8/11 <L1.start, L1.end> line up 0.3i with .start at 9/11 <L1.start, L1.end> line up 0.3i with .start at 10/11 <L1.start, L1.end> L2: line right 1i with .start at L1.start - (0, 0.5i) line up 0.3i with .start at 1/11 <L2.start, L2.end> line up 0.3i with .start at 2/11 <L2.start, L2.end> line up 0.3i with .start at 3/11 <L2.start, L2.end> line up 0.3i with .start at 4/11 <L2.start, L2.end> line up 0.3i with .start at 5/11 <L2.start, L2.end> line up 0.3i with .start at 6/11 <L2.start, L2.end> line up 0.3i with .start at 7/11 <L2.start, L2.end> line up 0.3i with .start at 8/11 <L2.start, L2.end> line up 0.3i with .start at 9/11 <L2.start, L2.end> line up 0.3i with .start at 10/11 <L2.start, L2.end> L3: line right 1i with .start at L2.start - (0, 0.5i) line up 0.3i with with .start at 3/11 <L3.start, L3.end> line up 0.3i with .start at 4/11 <L3.start, L3.end> line up 0.3i with .start at 5/11 <L3.start, L3.end> line up 0.3i with .start at 6/11 <L3.start, L3.end> line up 0.3i with .start at 7/11 <L3.start, L3.end> line up 0.3i with .start at 8/11 <L3.start, L3.end> line up 0.3i with .start at 9/11 <L3.start, L3.end> line up 0.3i with .start at 10/11 <L3.start, L3.end> L4: line right 1i with .start at L3.start - (0, 0.5i) line up 0.3i with .start at 1/11 <L4.start, L4.end> line up 0.3i with .start at 2/11 <L4.start, L4.end> line up 0.3i with .start at 3/11 <L4.start, L4.end> line up 0.3i with .start at 4/11 <L4.start, L4.end> line up 0.3i with .start at 5/11 <L4.start, L4.end> line up 0.3i with .start at 6/11 <L4.start, L4.end> line up 0.3i with with .start at 9/11 <L4.start, L4.end> line up 0.3i with .start at 10/11 <L4.start, L4.end> line right 0.3i at B.e move up 0.5i line down 1i arc to last line.end + (0.15i, -0.15i) rad 0.15i BOTTOM: line right 1.5i arc invis to last line.end + (0.15i, 0.15i) rad 0.15i line invis up 0.35i line up 0.15i RIGHT: line up 0.5i arc to last line.end - (0.15i, -0.15i) rad 0.15i TOP: line left 1.5i arc to last line.end - (0.15i, 0.15i) rad 0.15i define workstation X line up 0.15i line right 0.15i line up 0.30i line left 0.15i line down 0.20i line left 0.15i line down 0.10i line right 0.15i X move to TOP.end + (0.15i, 0) workstation move to TOP.center workstation move to TOP.start - (0.15i, 0) workstation "Workstations" at TOP.center + (0, 0.45i) above oldwid=boxwid oldht=boxht boxwid = 0.3i boxht = 0.3i line down 0.15i at BOTTOM.start + (0.15i, 0) box with .n at last line.end line down 0.15i at BOTTOM.center box with .n at last line.end line down 0.15i at BOTTOM.end - (0.15i, 0) X: box with .n at last line.end box invis "Specialized servers" "(file, data base, etc)" with .w at X.e + (0.6i, 0) line right 0.3i at RIGHT.start box arrow right 0.5i " WAN" ljust "Gateway" at last box.n above boxwid=oldwid boxht=oldht

Fig. 1. The Amoeba architecture.

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). 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 (although see Tanenbaum and van Renesse, 1988).

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.

boxht=0.5i A:box wid 2i "Server" "port" B:box wid 1i"Object" "number" C:box wid 0.5i "Rights" "" D:box wid 2i"Check" "field" "48" at A.n above "24" at B.n above "8" at C.n above "48" at D.n above

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.

2.3. Remote Operations

The combination of a request from a client to a server and a reply from a server to a client is called a *remote operation*. The request and reply messages consist of a header and a buffer. Headers are 32 bytes, and buffers can be up to 30 kilobytes. A request header contains the capability of the object to be operated on, the operation code, and a limited area (8 bytes) for parameters to the operation. For example, in a write operation on a file, the capability identifies the file, the operation code is *write*, and the parameters specify the size of the data to be written, and the offset in the file. The request buffer contains the data to be written. A reply header contains an error code, a limited area for the result of the operation (8 bytes), and a capability field that can be used to return a capability (e.g., as the result of the creation of an object, or of a directory search operation).

The primitives for doing remote operations are listed below:

get_request(req-header, req-buffer, req-size)
put_reply(rep-header, rep-buffer, rep-size)
do_operation(req-header, req-buffer, req-size, rep-header, rep-buffer, rep-size)

When a server is prepared to accept requests from clients, it executes a *get_request* primitive, which causes it to block. When a request message arrives, the server is unblocked and the formal parameters of the call to *get_request* are filled in with information from the incoming request. The server than performs the work and sends a reply using *put_reply*.

On the client side, to invoke a remote operation, a process uses *do_operation*. This action causes the request message to be sent to the server. The request header contains the capability of the object to be manipulated and various parameters relating to the operation. The caller is blocked until the reply is received, at which time the three rep- parameters are filled in and a status returned. The return status of *do_operation* can be one of three possibilities:

1. The request was delivered and has been executed.
2. The request was not delivered or executed (e.g., server was down).

3. The status is unknown.

The third case can arise when the request was sent (and possibly even acknowledged), but no reply was forthcoming. This situation can arise if a server crashes part way through the remote operation. Under all conditions of lost messages and crashed servers, Amoeba guarantees that messages are delivered at most once. If status 3 is returned, it is up to the application or run time system to do its own fault recovery.

2.4. Remote Procedure Calls

A remote procedure call actually consists of more than just the request/reply exchange described above. The client has to place the capability, operation code, and parameters in the request buffer, and on receiving the reply it has to unpack the results. The server has to check the capability, extract the operation code, and parameters from the request and call the appropriate procedure. The result of the procedure has to be placed in the reply buffer. Placing parameters or results in a message buffer is called *marshalling*, and has a non-trivial cost. Different data representations in client and server also have to be handled. All of these steps must be very carefully designed and coded, lest they introduce unacceptable overhead.

To hide the marshalling and message passing from the users, Amoeba uses *stub routines* (Birrell and Nelson, 1984). For example, one of the file system stubs might start with:

```
int read_file(file_cap, offset, nbytes, buffer)
        capability_t    *file_cap;
        long            offset;
        long            *nbytes;
        char            *buffer;
```

This call reads *nbytes* starting at *offset* from the file identified by *file_cap* into *buffer*. It returns the number of bytes read or an error number. A hand-written stub for this code is simple to construct: it will produce a request header containing *file_cap*, the operation code for *read_file*, *offset*, and *nbytes*, and invoke the remote operation:

```
do_operation(req_header, NULL, 0, repader, buffer, nbytes);
```

Automatic generation of such a stub from the procedure header above is impossible. Some essential information is missing. The author of the handwritten stub used several pieces of derived information to do the job.

1. The buffer is used only to receive information from the file server; it is an output parameter, and should not be sent to the server.
2. The maximum length of the buffer is given in the *nbytes* parameter. The actual length of the buffer is the returned value if there is no error and zero otherwise.
3. *File_cap* is special; it defines the service that must carry out the remote operation.

4. The stub generator does not know what the server's operation code for *read_file* is. This requires extra information. But, to be fair, the human stub writer needs this extra information too.

In order to be able to do automatic stub generation, the interfaces between client and servers have to contain the information listed above, plus information about type representation for all language/machine combinations used. In addition, the interface specifications have to have an *inheritance* mechanism which allows a lower-level interface to be shared by several other interfaces. The *read_file* operation, for instance, will be defined in a low-level interface which is then inherited by all file-server interfaces, the terminal-server interface, and the segment-server interface.

AIL (Amoeba Interface Language) is a language in which the extra information for the generation of efficient stubs can be specified, so that the AIL compiler can produce stub routines automatically (van Rossum, 1989). The *read_file* operation could be part of an interface (called *class* in AIL) whose definition could look something like

```
class simple_file_server [1000..1999] {
        read_file(*, in unsigned offset, in out unsigned nbytes,
                out char buffer[nbytes:NBYTES]);
        write_file(*, . . . );
};
```

From this specification, AIL can generate the C client stub of the example above with the correct marshalling code. The AIL specification tells AIL that the operation codes for the simple_file_server can be allocated in the range 1000 to 1999; it tells which parameters are input parameters to the server and which are output parameters from the server, and it tells that the length of buffer is at most *NBYTES* (which may be a constant or a variable) and that the actual length is *nbytes*.

The Bullet File Server, one of the file servers operational in Amoeba, *inherits* this interface, making it part of the Bullet File Server interface:

```
class bullet_server [2000..2999] {
        inherit simple_file_server;
        creat_file(*, . . . );
};
```

AIL can do *multiple inheritance* so the Bullet server interface can inherit both the simple file interface and a *capability management* interface, for instance, for restricting rights on capabilities.

2.5. Threads

A process in Amoeba consists of one or more threads that run in parallel. All the threads of a process share the same address space, but each one has a dedicated portion of that address space for use as its private stack, and each one has its own program counter. From the programmer's point of view, each thread is like a traditional sequential process, except that the threads of a process can communicate using shared memory. In addition, the threads can synchronize with each other using semaphores.

The purpose of having multiple threads in a process is to increase performance through parallelism, and still provide a reasonable semantic model to the programmer. For example, a file server could be programmed as a process with multiple threads. When a request comes in, it can be given to some thread to handle. That thread first checks an internal (software) cache to see if the needed data are present. If not, it performs an RPC with a remote disk server to acquire the data.

While waiting for the reply from the disk, the thread is blocked and will not be able to handle any other requests. However, new requests can be given to other threads in the same process to work on while the first thread is blocked. In this way, multiple requests can be handled simultaneously, while allowing each thread to work in a sequential way. The point of having all the threads share a common address space is to make it possible for all of them to have direct access to a common cache, something that is not possible if each thread is its own address space.

The scheduling of threads within a process is done by code within the process itself. When a thread blocks, either because it has no work to do (i.e., on a *get_request*) or because it is waiting for a remote reply (i.e., on a *do_operation*), the internal scheduler is called, the thread is blocked, and a new thread can be run. Threads are not pre-empted, that is, the currently running thread will not be stopped because it has run too long. This decision was made to avoid race conditions. A thread need not worry that when it is halfway through updating some critical shared table it will be suddenly stopped and some other thread will start up and try to use the table. It is assumed that the threads in a process were all written by the same programmer and are actively co-operating. That is why they are in the same process. Thus the interaction between two threads in the same process is quite different than the interaction between two threads in different processes, which may be hostile to one another and for which hardware memory protection is required and used.

3. SERVERS

The Amoeba kernel, as described above, 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 processes. Even capability management is done entirely in user space, since the cryptographic technique discussed earlier makes it virtually impossible for users to generate counterfeit capabilities.

All of the remaining functions that are normally associated with a modern operating system environment are performed by servers, which are just ordinary user processes. The file system, for example, consists of a collection of user processes. Any user who is not happy with the standard file system is free to write and use his own. This situation can be contrasted with a system like UNIX,† in which there is a single file system that all applications must use, no matter how

† UNIX is a trademark of Bell Laboratories.

inappropriate it may be. (Stonebraker, 1981) for example, discusses the numerous problems that UNIX creates for database systems.

In the following sections we will discuss the Amoeba memory server, process server, device servers, file server, directory server, and boot server as examples of typical Amoeba servers. Many others exist as well.

3.1. The Memory Server

In many applications, processes need a way to create subprocesses. In UNIX, a subprocess is created by the *fork* primitive, in which an exact copy of of the original process is made. This process can then run for a while, attending to housekeeping activities, and then issue an *exec* primitive to overwrite its core image with a new program.

In a distributed system, this model is not attractive. The idea of first building an exact copy of the process, possibly remotely, and then throwing it away again shortly thereafter is inefficient. Consequently, Amoeba uses a different strategy. Each Amoeba machine runs a memory server process, whose job is to manage memory. It offers primitives to allocate, free, read and write chunks of memory called *segments*. These primitives, together with those of the Process Server below, are used instead of UNIX's *fork* and *exec*.

3.2. The Process Server

A typical scenario for creating a new process is as follows. The process wanting to create a subprocess first does an RPC with the process server telling it that it wants to create a process, specifying the information about the new process, such as machine requirements (e.g., is floating point hardware needed, how much memory is required) and the name of the program to be run. The process server then chooses a machine to run the new process on, based on its information about system load, location of input data, and other relevant factors that might affect performance.

Once the process server has made a choice, it contacts the memory server on the chosen machine. It then asks the memory server to create segments for the text, data, and stack of the new process, and possibly other segments, as needed. For each segment it creates, the memory server returns a capability to the process server. The process server then uses these capabilities to perform write operations, that is, to fill the segments with the initial code, data, and stack values.

When all the segments have been loaded, the process server asks the memory server to execute a *build-process* operation, with the segment capabilities as the input parameters. The memory server responds by returning a capability for a newly-minted process. The process server can then issue an *execute* operation using this capability to start the new process going. When the process terminates, it returns a status value to the process server.

3.3. The File Server

As far as the system is concerned, a file server is just another user process. Consequently, a variety of file servers have been written for Amoeba in the course of its existence. The first one, *FUSS* (*Free University Storage System*) (Mullender and Tanenbaum, 1985) was designed as an experiment in managing concurrent access using optimistic concurrency control. The second one was designed for UNIX emulation, and is currently heavily used. The third one, the *bullet server* was designed for extremely high performance (van Renesse et al., 1988, 1989a, 1989b). It is this one that we will describe below.

The decrease in the cost of disk and RAM memories over the past decade has allowed to use a radically different design than is used in UNIX and most other operating systems. In particular, we have abandoned the idea of storing files as a collection of fixed size disk blocks. All files are stored contiguously, both on the disk and in the server's main memory. While this design wastes some disk space and memory due to fragmentation overhead, we feel that the enormous gain in performance (described in section 6) more than offsets the small extra cost of having to buy, say, an 800 MB disk instead of a 500 MB diskin order to store 500 MB worth of files.

The bullet[A server is an immutable file store, with as principal operations *read-file* and *create-file*. (For garbage collection purposes there is also a *delete-file* operation.) When a process issues a *read-file* request, the bullet server can transfer the entire file to the client in a single RPC, unless it is larger than the maximum size (30K), in which case multiple RPCs are needed. The client can then edit or otherwise modify the file locally. When it is finished, the client issues a *create-file* RPC to make a new version. The old version remains intact until explicitly deleted or garbage collected. Note that different versions of a file have different capabilities, so they can co-exist, making it simple to implement source code control systems.

The files are stored contiguously on disk, and are cached in memory (currently 16 Mbytes). When a requested file is not available in this memory, it is loaded from disk in a single large DMA operation and stored contiguously in the cache. (Unlike conventional file systems, there are no ‘‘blocks’’ used anywhere in the file system.) In the *create-file* operation one can request the reply before the file is written to disk (for speed), or afterwards (to know that it has been successfully written).

When the bullet server is booted, the entire ‘‘i-node table’’ is read into memory in a single disk operation and kept there while the server is running. When a file operation is requested, the object number field in the capability is extracted, which is an index into this table. The entry thus located gives the disk address as well as the cache address of the contiguous file (if present). No disk access is needed to fetch the ‘‘i-node’’ and at most one disk access is needed to fetch the file itself, if it is not in the cache. The simplicity of this design trades off some space for very high performance. We will discuss the performance of this server in Section 6.

3.4. The Directory Server

The bullet server does not provide any naming services. To access a file, a process must provide the relevant capability. Since working with 128-bit binary numbers is not convenient for people, we have designed and implemented a directory server to manage names and capabilities.

The directory server manages multiple directories, each of which is a normal object. Stripped down to its barest essentials, a directory maps ASCII strings onto a capabilities. A process can present a string, such as a file name, to the directory server, and the directory server returns the capability for that file. Using this capability, the process can then access the file.

In UNIX terms, when a file is opened, the capability is retrieved from the directory server for use in subsequent read and write operations. After the capability has been fetched from the directory server, subsequent RPCs go directly to the server that manages the object. The directory server is no longer involved.

It is important to realize that the directory server simply provides a mapping function. The client provides a capability for a directory (in order to specify which directory to search) and a string, and the directory server looks up the string in the specified directory and returns the capability associated with the string. The directory server has no knowledge of the kind of object that the capability controls.

In particular, it can be a capability for another directory on the same or a different directory server, a file, a mailbox, a database, a process capability, a segment capability, a capability for a piece of hardware, or anything else. Furthermore, the capability may be for an object located on the same machine, a different machine on the local network, or a capability for an object in a foreign country. The nature and location of the object is completely arbitrary. Thus the objects in a directory need not all be on the same disk, for example, as is the case in many systems that support ‘‘remote mount’’ operations.

Since a directory may contain entries for other directories, it is possible to build up arbitrary directory structures, including trees and graphs. As an optimization, it is possible to give the directory server a complete path, and have it follow it as far as it can, returning a single capability at the end.

Actually, directories are slightly more general than just simple mappings. It is commonly the case that the owner of a file may want to have the right to perform all operations on it, but may want to permit others read-only access. The directory server supports this idea by structuring directories as a series of rows, one per object, as shown in Fig. 3

The first column gives the string (e.g., the file name). The second column gives the capability that goes with that string. The remaining columns each apply to one user class. For example, one could set up a directory with different access rights for the owner, the owner’s group, and others, as in UNIX, but other combinations are also possible.

The capability for a directory specifies the columns to which the holder has access as a bit map in part of the rights field (e.g., 3 bits). Thus in the above example, the bits 001 might specify access to only the *Other* column. In Sec. 2.2 we discussed how the rights bits are

Object name	Capability	Owner	Group	Other
.	cap1	11111	11000	10000
games_dir	cap2	11111	10000	10000
paper.t	cap3	11111	00000	00000
prog.c	cap4	11111	11100	10000

Fig. 3. A directory with three user classes, four entries, and five rights.

protected from tampering by use of the check field.

To understand how the use of multiple columns works, let us consider a typical access. The client provides a capability for a directory, a string, and a column. The string is looked up in the directory to find the proper row. Next, the column is checked against the (singleton) bit map in the rights field, to see which column should be used. Remember that the cryptographic scheme described in Sec. 2.2 prevents users from modifying the bit map, hence accessing a forbidden column.

Then the entry in the selected row and column is extracted. Conceptually this is just a capability, with the proper rights bits turned on. However, to avoid having to store many capabilities, few of which are ever used, an optimization is made, and the entry is just a bit map, *b*. The directory server can then ask the server that manages the object to return a new capability with only those rights in *b*. This new capability is returned to the user and also cached for future use, to reduce calls to the server.

The directory server supports a number of operations on directory objects. These including looking up capabilities, adding new rows to a directory, removing rows from directories, listing directories, inquiring about the status of directories and objects, and deleting directories. There is also provision for performing multiple operations in a single atomic action, to provide for fault tolerance.

Furthermore, there is also support of handling replicated objects. The capability field in Fig. 3 can actually hold a set of capabilities for multiple copies of each object. Thus when a process looks up an object, it can retrieve the entire set of capabilities for all the copies. If one of the objects is unavailable, the other ones can be tried. In addition, when a new object is installed in a directory, an option is available to have the directory server itself request copies to be made, and then store all the capabilities, thus freeing the user from this administration.

4. WIDE-AREA AMOEBA

Amoeba was designed with the idea that a collection of machines on a local network would be able to communicate over a wide-area network with a similar collection of remote machines. The key problem here is that wide-area networks are slow and unreliable, and furthermore use protocols such as X.25, TCP/IP, and OSI, in any event, not RPC. The primary goal of the wide-area networking in Amoeba has been to achieve transparency without sacrificing performance.

In particular, it is undesirable that the very fast local RPC be slowed down in any way due to the existence of wide-area communication. We believe this goal has been achieved.

The Amoeba world is divided up into *domains*, each domain being an interconnected collection of local area networks. The key aspect of a domain (e.g., a campus), is that broadcasts done from any machine in the domain are received by all other machines in the domain, but not by machines outside the domain.

The importance of broadcasting has to do with how ports are located in Amoeba. When a process does an RPC with a port not previously used, the kernel broadcasts a *locate* message. The server responds to this broadcast with its address, which is then used and also cached for future RPCs.

This strategy is undesirable with a wide-area network. Although broadcast can be simulated using a minimum spanning tree (Delal, 1977) it is expensive and inefficient. Furthermore, not every service should be available worldwide. For example, a laser printer server on the third floor of the physics building at a university in California may not be of much use to clients in New York.

Both of these problems are dealt with by introducing the concept of *publishing*. When a service wishes to be known and accessible outside its own domain, it contacts the *Service for Wide-Area Networks* (SWAN) and asks that its port be published in some set of domains. The SWAN publishes the port by doing RPCs with SWAN processes in each of those domains.

When a port is published in a domain, a new process called a *server agent* is created in that domain. The process typically runs on the gateway machine, and does a *get_request* using the remote server's port. It is quiescent until its server is needed, at which time it comes to life and performs an RPC with the server.

Now let us consider what happens when a process tries to locate a remote server whose port has been published. The process' kernel broadcasts a *locate*, which is received by the server agent. The client agent then builds a message and hands it to a *link* process on the gateway machine. The link process forwards it over the wide-area network to the server's domain, where it arrives at the gateway, causing a *client agent* process to be created. This client agent then makes a normal RPC to the server. The set of processes involved here is shown in Fig. 4

C: box move G1: box line 1.5i "Wide-area network" below G2: box move S: box LAN1: line boxwid+boxwid+movewid from C.sw - (0i, 0.5i) LAN2: line boxwid+boxwid+movewid from G2.sw - (0i, 0.5i) line down 0.5i from C.s line down 0.5i from G1.s line down 0.5i from G2.s line down 0.5i from S.s "Client" at C.n above "Gateway" at G1.n above "Gateway" at G2.n above "Server" at S.n above "LAN 1" at LAN1.s + (boxwid + movewid/2, -0.1i) below "LAN 2" at LAN2.s + (boxwid + movewid/2, -0.1i) below circle "C" at C.c radius boxht/5 circle "S" at S.c radius boxht/5 circle "SA" at G1.c - (0.15i, 0) radius boxht/5 circle "L" at G1.c + (0.15i, 0) radius boxht/5 circle "L" at G2.c - (0.15i, 0) radius boxht/5 circle "CA" at G2.c + (0.15i, 0) radius boxht/5

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

The beauty of this scheme is that it is completely transparent. Neither user processes nor the kernel know which processes are local and which are remote. The communication between the client and the server agent is completely local, using the normal RPC. Similarly, the communication between the client agent and the server is also completely normal. Neither the client nor the server knows that it is talking to a distant process.

Of course, the two agents are well aware of what is going on, but they are automatically generated as needed, and are not visible to users. The link processes are the only ones that know about the details of the wide-area network. They talk to the agents using RPC, but to each other using whatever protocol the wide-area network requires. The point of splitting off the agents from the link processes is to completely isolate the technical details of the wide-area network in one kind of process, and to make it easier to have multiway gateways, which would have one type of link process for each wide-area network type to which the gateway is attached.

It is important to note that this design causes no performance degradation whatsoever for local communication. An RPC between a client and a server on the same LAN proceeds at full speed, with no relaying of any kind. Clearly there is some performance loss when a client is talking to a server located on a distant network, but the limiting factor is invariably the bandwidth of the wide-area network, so the extra overhead of having messages being relayed several times is negligible.

Another useful aspect of this design is the management controls it allows. To start with, services can only be published with the help of the SWAN server, which can check to see if the system administration wants the port be to published. Another important control is the ability to prevent certain processes (e.g., those owned by students) from accessing wide-area services, since all such traffic must pass through the gateways, and various checks can be made there. Finally, the gateways can do accounting, statistics gathering, and monitoring of the wide-area network.

5. APPLICATIONS

Amoeba has been used to program a variety of applications. In this section we will describe several of them, including UNIX emulation, parallel make, traveling salesman, and alpha-beta search.

5.1. UNIX Emulation

One of the goals of Amoeba was to make it useful as a program development environment. For such an environment, one needs editors, compilers, and numerous other standard software. It was decided that the easiest way to obtain this software was to emulate UNIX and then to run UNIX and MINIX (Tanenbaum, 1977) software on top of it.

The Amoeba 3.0 UNIX emulation was done mostly by a UNIX Version 7 compatible file server (derived from the Minix file server). This file server accepted request messages asking for system calls such as *read* and *write*, and carried them out, returning the results in reply messages. In addition, there is another server that handles those V7 system calls that do not relate to the file system, such as *fork* and *exec*.

Using a special set of library procedures that do RPCs with these servers, it has been possible to construct an emulation of the UNIX system call interface that is good enough that about 100 of the most common utility programs have been ported to Amoeba. The Amoeba user can now use most of the standard editors, compilers, file utilities and other programs in a way that looks very much like UNIX, although in fact it is really Amoeba.

5.2. Parallel Make

As shown in Figure 1, the hardware on which Amoeba runs contains a processor pool with several dozen 68020 and 68030 processors. One obvious application for these processors in a UNIX environment is a parallel version of *make* (Feldman, 1985). The idea here is that when *make* discovers that multiple compilations are needed, they are run in parallel on different processors.

Although this idea sounds simple, there are several potential problems. For one, to make a single target file, a sequence of several commands may have to be executed, and some of these may use files created by earlier ones. The solution chosen is to let each command execute in parallel, but block when it needs a file not yet available.

Other problems relate to technical limitations of the *make* program. For example, since it expects commands to be run sequentially, rather than in parallel, it does not keep track of how many processes it has forked off, which may exceed various system limits.

Finally, there are programs, such as *yacc* (Johnson, 1978) that write their output on fixed name files, such as *y.tab.c*. When multiple *yacc*s are running in the same directory, they all write to the same file, thus producing gibberish. All of these problems have been dealt with by one means or another, as described in (Baalbergen, 1988).

The parallel compilations are directed by a new version of *make*, called *pmake*, based on the UNIX one but with additional code to handle parallelism. The *makefiles* accepted by this program are compatible with the standard one.

The performance of *pmake* depends strongly on the input. When making a program consisting of many medium-sized files, considerable speedup can be achieved. However, when a program has one large source file and many small ones, the total time can never be smaller than the compilation time of the large one. Furthermore, the time required by *pmake* itself cannot be neglected. All in all, a speedup of about a factor of 4 over sequential *make* has been observed in practice on typical *makefiles*.

5.3. The Traveling Salesman Problem

In addition to various experiments with the UNIX software, we have also tried programming some applications in parallel. Typical applications are the traveling salesman problem (Lawler and Wood, 1966) and alpha-beta search (Marsland and Campbel, 1982) We briefly describe these below. More details can be found in (van Renesse and Tanenbaum, 1987).

In the traveling salesman problem, the computer is given a starting location and a list of cities to be visited. The idea is to find the shortest path that visits each city exactly once, and then returns to the starting place. Using Amoeba we have programmed this application in parallel by having one pool processor act as coordinator, and the rest as slaves.

Suppose, for example, that the starting place is London, and the cities to be visited include New York, Sydney, Nairobi, and Tokyo. The coordinator might tell the first slave to investigate all paths starting with London-New York, the second slave to investigate all paths starting with London-Sydney, the third slave to investigate all paths starting with London-Nairobi, and so on. All of these searches go on in parallel. When a slave is finished, it reports back to the coordinator and gets a new assignment.

The algorithm can be applied recursively. For example, the first slave could allocate a processor to investigate paths starting with London-New York-Sydney, another processor to investigate London-New York-Nairobi, and so forth. At some point, of course, a cutoff is needed at which a slave actually does the calculation itself and does not try to farm it out to other processors.

The performance of the algorithm can be greatly improved by keeping track of the best total path found so far. A good initial path can be found by using the ‘‘closest city next’’ heuristic. Whenever a slave is started up, it is given the length of the best total path so far. If it ever finds itself working on a partial path that is longer than the best-known total path, it immediately stops what it is doing, reports back failure, and asks for more work. Initial experiments have shown that about 75 percent of the theoretical maximum speedup can be achieved using this algorithm, the remaining 1/4 being lost to communication and other overhead. More recent results suggest that this can be improved considerably.

5.4. Alpha-Beta Search

Another application that we have programmed in parallel using Amoeba is game playing using the alpha-beta heuristic for pruning the search tree. The general idea is the same as for the traveling salesman. When a processor is given a board to evaluate, it generates all the legal moves possible starting at that board, and hands them off to others to evaluate in parallel.

The alpha-beta heuristic is commonly used in two-person, zero-sum games to prune the search tree. A window of values is established, and positions that fall outside this window are not examined because better moves are known to exist. In contrast to the traveling salesman problem, in which much of the tree has to be searched, alpha-beta allows a much greater pruning if the positions are evaluated in a well chosen order.

For example, on a single machine, we might have three legal moves A, B, and C at some point. As a result of evaluating A we might discover that looking at its siblings in the tree, B and C was pointless. In a parallel implementation, we would do all at once, and ultimately waste the computing power devoted to B and C. The result is that much parallel searching is wasted, and the net result is not that much better than a sequential algorithm on a single processor. Our experiments running Othello (Reversi) on Amoeba have shown that we were unable to utilize more than 40 percent of the total processor capacity available, compared to 75 percent for the traveling salesman problem. Work is in progress to improve this result.

6. PERFORMANCE

Amoeba was designed to be very fast. Measurements show that this goal has been achieved. In this Section, we will present the results of some timing experiments we have done. These measurements were performed on 16 MHz Motorola 68020 processors (Tadpole VME boards) running Amoeba on the bare hardware (no UNIX), and for comparison purposes, on SUN 3/50 workstations running SUN OS 3.5 UNIX to whose kernel the Amoeba driver was added. All processors were connected over a 10 Mbps Ethernet using LANCE chip interfaces. We measured the performance for three different configurations:

1. Two user processes running on Amoeba.
2. Two user processes running on SUN UNIX but using the Amoeba primitives.
3. Two user processes running on SUN UNIX and using SUN RPC.

The latter two were for comparison purposes only. We ran tests for the local case (both processes on the same machine) and for the remote case (each process on a separate machine, with communication over the Ethernet). In all cases communication was from process to process, all of which were running in user mode outside the kernel.

For each configuration (pure Amoeba, Amoeba primitives on UNIX, SUN RPC on UNIX), we tried to run three test cases: a 4-byte message (1 integer), an 8 Kbyte message, and a 30 Kbyte message. The 4-byte message test is typical for short control messages, the 8-Kbyte message is typical for reading a medium-sized file from a remote file, and the 30-Kbyte test is the maximum the current implementation of Amoeba can handle. Thus, in total we should have 9 cases (3 configurations and 3 sizes). However, the standard SUN RPC is limited to 8K, so we have measurements for only eight of them.

In Fig. 5 we give the delay and the bandwidth of these eight cases, both for local processes (same machine) and remote processes (different machines). The delay is the time as seen from the client, running as a user process, between the calling of and returning from the RPC primitive. The bandwidth is the number of data bytes per second that the client receives from the server, excluding headers. The measurements were done for both local RPCs, where the client and server processes were running on the same processor, and for truly remote RPCs.

The interesting comparisons in these tables are the comparisons of pure Amoeba and pure SUN UNIX both for short communications, where delay is critical, and long ones, where bandwidth is the issue. A 4-byte Amoeba RPC takes 1.4 msec, vs. 12.2 msec for SUN RPC. Similarly, for 8 Kbyte RPCs, the Amoeba bandwidth is 625 Kbytes/sec, vs. only 202 Kbytes for the SUN RPC. The conclusion is that Amoeba's delay is 9 times better and its throughput is 3 times better. (For the record, we should note that the Amoeba 68020s ran at 16 MHz vs. 15 MHz for the SUNs, but this only changes the results slightly.)

	Delay (msec)			Bandwidth (Kbytes/sec)		
	case 1 *(4 bytes)*	case 2 *(8 Kb)*	case 3 *(30 Kb)*	case 1 *(4 bytes)*	case 2 *(8 Kb)*	case 3 *(30 Kb)*
pure Amoeba local	0.8	2.5	7.1	5.0	3277	4255
pure Amoeba remote	1.4	13.1	44.0	2.9	625	677
UNIX driver local	4.5	10.0	32.0	0.9	819	938
UNIX driver remote	7.0	36.4	134.0	0.6	225	224
SUN RPC local	10.4	23.6	imposs.	0.4	347	imposs.
SUN RPC remote	12.2	40.6	imposs.	0.3	202	imposs.
		(a)			(b)	

Fig. 5. RPC between user processes in three common cases for three different systems. Local RPCs are RPCs where the client and server are running on the same processor. (a) Delay in msec. (b) Bandwidth in Kbytes/sec. The UNIX driver implements Amoeba RPCs and Amoeba protocol under SUN UNIX.

While the SUN is obviously not the only system of interest, its widespread use and excellent performance makes it a convenient benchmark. We have looked in the literature for performance figures from other distributed systems and have asked many other researchers, and to the best of our knowledge, no other operating system has a lower RPC delay or higher bandwidth on this class of hardware. In particular, our tests measure delay and throughput from user process to user process (not kernel to kernel) and do not involve any tricks or special cases.

Noteworthy are the performance of the V-system (Cheriton, 1988) and the Firefly RPC (Burrows and Schroeder, 1987), V because it was also designed for high performance, and the Firefly because it is a multiprocessor. For V we find that delay for null RPCs is 2.54 msec (Cheriton, 1988) (vs. 1.4 msec for Amoeba), and that the bandwidth for 8 Kbyte RPCs is 460 Kbytes/sec (vs. 625 Kbytes/sec for Amoeba). For larger requests (up to 16 Kbytes), the data rate increases to 550 Kbytes/sec (vs. 644 Kbytes/sec for Amoeba). All of these figures represent a performance considerably worse than that of Amoeba, despite the fact that the V measurements were made on substantially faster hardware, namely SUN 3/75s (vs. SUN 3/50s for Amoeba).

The Firefly is an experimental multi-processor under development at DEC SRC. A Firefly contains five MicroVax CPUs, each of which has about half the computing power of the SUN 3/50, but which collectively have much more. The null RPC time has been clocked at 2.66 msec. The bandwidth for one client and one server is 228 Kbytes/sec. Using four threads in the client, the bandwidth can be increased to 582 Kbytes/sec. To be able to achieve this good performance, the RPC subsystem has been carefully coded in VAX assembly language. In contrast, the Amoeba RPC code is written entirely in C, with no assembly code. Carefully recoding it in assembler would no doubt give a considerable gain in performance.

Like Amoeba itself, the bullet server was designed for extremely high performance. Below we present some measurements of what has been achieved. Figure 6 gives the performance of the bullet server for tests made with files of 1 Kbyte, 16 Kbytes, and 1 Mbyte. In the first column the delay and bandwidth for read operations is shown. Note that the test file will be completely in memory, and no disk access is necessary. In the second column a create and a delete operation together is measured, and the file is written to two disks (to provide fault tolerance and stable storage). Note that both operations involve disk requests. Moreover, the create operation has to generate a capability, which involves costly operations such as generating a random number and encrypting it using a one-way function based on DES. These operations alone account for a significant amount of time.

	Delay (msec)	
File Size	READ	CREATE+DEL
1 Kbyte	3	130
16 Kbyte	25	168
1 Mbyte	1550	4160

(a)

	Bandwidth (Kbytes/sec)	
File Size	READ	CREATE+DEL
1 Kbyte	341	7
16 Kbyte	650	98
1 Mbyte	677	379

(b)

Fig. 6. Performance of the Bullet file server for read operations, and create and delete operations together. (a) Delay in msec. (b) Bandwidth in Kbytes/sec.

To compare this with the SUN NFS file system, we have measured reading and creating files on a SUN 3/50 using a remote SUN 3/180 file server (using 16.7 MHz 68020s and SUN OS 3.5), equipped with a 3 Mbyte buffer cache. The measurements were made on an idle system. To disable local caching on the SUN 3/50, we have locked the file using the SUN UNIX *lockf* primitive. The read test consists of an *lseek* followed by a *read* system call. The write test consists of consecutively executing *creat*, *write*, and *close*. The SUN NFS file server uses a write-through cache, but writes the file to one disk only. The results are depicted in Fig. 7.

	Delay (msec)	
File Size	READ	CREATE
1 Kbyte	10	97
16 Kbyte	47	191
1 Mbyte	3345	15,850

(a)

	Bandwidth (Kbytes/sec)	
File Size	READ	CREATE
1 Kbyte	98	11
16 Kbyte	349	86
1 Mbyte	313	66

(b)

Fig. 7. Performance of the SUN NFS file server for read and create operations. (a) Delay in msec. (b) Bandwidth in Kbytes/sec.

Observe that reading and creating 1 Mbyte files result in lower bandwidths than reading and creating 16 Kbyte files. The Bullet file server's performance for read operations is two to three times better than the SUN NFS file server. For create operations, the Bullet file server has a constant overhead for producing and encrypting capabilities. For small files we therefore observe a lower bandwidth than for SUN NFS. Although the Bullet file server replicates its files on two disks, for writing large files, the bandwidth is nevertheless four times that of SUN NFS.

7. EVALUATION

In this section we will take a critical look at Amoeba and its evolution and point out some aspects that we consider successful and others that we consider less successful. In areas where Amoeba 3.0 was found wanting, we have made improvements in Amoeba 4.0, which is currently under development. These improvements are discussed below.

One area where little improvement is needed is portability. Amoeba started out on the 680x0 CPUs, and has been easily moved to the VAX, NS 32016 and Intel 80386. The Amoeba RPC protocol also has been implemented as part of MINIX V1.3, and as such is in widespread use around the world.

7.1. Objects and Capabilities

On the whole, the basic ideas of an object-based system has worked well. It has given us a framework which makes it easy to think about the system. When new objects or services are proposed, we have a clear model to deal with and specific questions to answer. In particular, for each new service, we must decide what objects will be supported and what operations will be permitted on these objects. This structuring technique has been valuable on many occasions.

The use of capabilities for naming and protecting objects has also been a success. By using cryptographically protected capabilities, we have a unique system-wide fixed length name for each object, yielding a high degree of transparency. Thus it is simple to implement a basic directory as a set of (ASCII string, capability) pairs. As a result, a directory may contain names for many kinds of objects, located all over the world and windows can be written on by any process holding the appropriate capability, no matter where it is. We feel this model is conceptually both simpler and more flexible than models using remote mounting and symbolic links such as SUN's NFS. Furthermore, it can be implemented just as efficiently.

We are also satisfied with the low-level user primitives. In effect there are only three principal system calls, *get_request*, *put_reply*, and *do_operation*, each easy to understand. All communication is based on these primitives, which are much simpler than, for example the socket interface in Berkeley UNIX, with its myriad of system calls, parameters, and options.

Amoeba 4.0 uses 256-bit capabilites, rather than the 128-bit capabilities of Amoeba 3.0. The larger Check field is more secure against attack, and other security aspects have also been tightened, including the addition of secure, encrypted communication between client and server. Also, the larger capabilities now have room for a *location hint* which can be exploited by the SWAN servers for locating objects in the wide-area network. Third, all the fields of the new

256-bit capability are now all aligned at 32-bit boundaries which potentially may give better performance.

7.2. Remote Procedure Call

For the most part, RPC communication is satisfactory, but sometimes it gives problems (Tanenbaum and van Renesse, 1988). In particular, RPC is inherently master-slave and point-to-point. Sometimes both of these issues lead to problems. In a UNIX pipeline, such as:

pic file | eqn | tbl | troff >outfile

for example, there is no inherent master-slave relationship, and it is not at all obvious if data movement between the elements of the pipeline should be read driven or write driven. We are still experimenting with various approaches here.

RPC is also point-to-point, which gives problems in parallel applications like the traveling salesman problem. When a process discovers a path that is better than the best known current path, what it really wants to do is send a multicast message to a large number of processes to inform all of them immediately. At present this is impossible, and must either be simulated with multiple RPCs or designed around.

Amoeba 4.0 fully supports broadcasting and multicasting, integrated into the RPC mechanism. In addition to the usual *unicast ports*, Amoeba 4.0 also supports *multicast ports*. A message sent to a multicast port is delivered to all of them, or at least an attempt is made. A higher-level protocol has been devised to implement 100% reliable multicasting with very low overhead. This protocol will be the subject of a forthcoming paper.

7.3. Memory and Process Management

Probably the worst mistake in the design of the Amoeba 3.0 process management mechanisms was the decision to have threads run to completion, that is, not be pre-emptable. The idea was that once a thread starting using some critical table, it would not be interrupted by another thread in the same cluster until it logically blocked. This scheme seemed simple to understand, and it was certainly easy to program.

Problems arose because programmers did not have a very good concept of when a process blocked. For example, to debug some code in a critical region, a programmer might add some print statements in the middle of the critical region code. These print statements might call library procedures that performed RPCs with a remote terminal server. While blocked waiting for the acknowledgement, a thread could be interrupted, and another thread could access the critical region, wreaking havoc. Thus the sanctity of the critical region could be destroyed by putting in print statements. Needless to say, this property was very confusing to naive programmers. In Amoeba 4.0 a more explicit mechanism has been introduced for guarding critical region code.

The run-to-completion semantics of thread scheduling in Amoeba 3.0 also prevents a multiprocessor implementation from exploiting parallelism and shared memory by allocating different threads in one process to different processors. Amoeba 4.0 threads can be run in parallel. No promises are made by the scheduler about allowing a thread to run until it blocks before another thread is scheduled. Threads sharing resources must explicitly synchronize using the semaphores or mutexes that Amoeba 4.0 provides for the purpose.

For Amoeba 4.0, we have thoroughly redesigned process-management and memory-management mechanisms. Under the Amoeba 3.0 regime, when a process started a new process (on a different machine, usually), it had to fetch the code from the file system and send it to the new process' host machine. Code would thus typically get copied over the network twice. In Amoeba 3.0, there were virtually no facilities for debugging active processes and we considered the control that processes had over their address space insufficient.

The Amoeba 4.0 process-management and memory-management mechanisms were designed to make process creation, migration, checkpointing and debugging all simple operations. Two key notions form the basis of these mechanisms. The first is a data structure, called *process descriptor*, which describes the state of an active process. The second is the *memory segment*, an object consisting of an array of bytes in memory with a capability that can be read and written like a file and that can also form part of a process' address space by being mapped into it.

The idea of a process descriptor is that it describes a process *in limbo*, a process just before it starts to run, or a process being migrated from one machine to another, or a process suspended while being debugged. A process descriptor has four components. The first describes the requirements for the system where the process must run: the class of machines, which instruction set, minimum available memory, use of special instructions such as floating point, and several more. The second component describes the layout of the address space: number of segments and, for each segment, the size, the virtual address, how it is mapped (e.g., read only, read-write, code/data space), and the capability of a file or segment containing the contents of the segment. The third component describes the state of each thread of control: stack pointer, stack top and bottom, program counter, processor status word, and registers. Threads can be blocked on certain system calls (e.g., get_request, acquire_semaphore); this can also be described. The fourth component is a list of ports for which the process is a server. This list is helpful to the kernel when it comes to buffering incoming requests and replying to port-locate operations.

In Amoeba 4.0, to create a process, one needs to do the following.

1. Get the process descriptor for the binary from the file system (command interpreters can cache process descriptors for efficiency).
2. Create a local segment or a file and initialize it to the initial environment of the new process. The environment consists of a set of named capabilities (a primitive directory, as it were), and the arguments to the process (in Unix terms, *argc* and *argv*; for Unix processes, one also adds the *environment variables*).
3. Modify the process descriptor to make the first segment the environment segment just created.

4. Send the process descriptor to the new process' host.

The host then allocates memory for local segments, reads the remote segments into the local ones, initializes the required number of threads and starts the process.

To stop a process, one can send it a signal. The process is then stopped and a process descriptor is made which is then sent to the process' owner for debugging. The owner can examine the process, modify it and resume its execution, or kill it. Similar mechanisms are used for checkpointing or migration.

The new memory-management mechanisms allow code caching in pool processors. They also give processes more control over the management of their address space because they can map segments into it or out of it. In particular, the mechanism allows the implementation of memory-mapped file i/o, shared libraries, and dynamic linking.

The new process-management mechanism allows migration, checkpointing and debugging, code caching and process images need to be copied over the network only once at most.

Using the Amoeba 4.0 process-management facilities, we plan to implement algorithms for code caching on the pool processors in conjunction with services that attempt to place processes on pool processors in such a way as to minimize process-startup times. Also, we intend to implement shared libraries and dynamic linking.

7.4. File System

One area of the system which we think has been eminently successful 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.

The entire replication process takes place in the background (lazy replication), and is entirely automatic, thus not bothering the user at all. We regard the file system as the most innovative part of the Amoeba 3.0 design, combining extremely high performance with reliability, robustness, and ease of use. We have no plans to change it.

7.5. Internetworking

We are also happy with the way wide-area networking has been handled, using server agents, client agents, and the SWAN. In particular, the fact that the existence of wide-area networking does not affect the protocols or performance of local RPCs at all is crucial. Many other designs (e.g., TCP/IP, OSI) start out with the wide-area case, and then use this locally as well. This choice results in significantly lower performance on a LAN than the Amoeba design, and no better performance over wide-area networks.

One configuration that was not adequately dealt with in Amoeba 3.0 is a system consisting of a large number of local area networks interconnected by many bridges and gateways. Although Amoeba 3.0 works on these systems, its performance is poor, partly due to the way port location and message handling is done. In Amoeba 4.0, we have designed and implemented a completely new low-level protocol called the *Fast Local Internet Protocol* (FLIP), that will greatly improve the performance in complex internets. Among other features, entire messages are now acknowledged instead of individual packets, greatly reducing the number of interrupts that must be processed. Port location is also done more efficiently, and a single server agent can now listen to an arbitrary number of ports, enormously reducing the number of quiescent server agents required in large systems.

7.6. UNIX Emulation

The Amoeba 3.0 UNIX emulation primarily consists of having borrowed the MINIX file server. This was a quick and dirty solution, but it means that Amoeba 3.0 UNIX programs cannot use the bullet server.

In Amoeba 4.0, a more complete UNIX emulation is done through a library of procedures that emulate the UNIX system calls by making calls to the bullet server, directory server, etc. This library, called *ajax*, has made it possible to port a large number of the standard UNIX utility programs to Amoeba using the normal Amoeba file system.

7.7. Parallel Applications

Although Amoeba was originally conceived as a system for *distributed* computing, the existence of the processor pool with 40 or so 680x0 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. To program these parallel applications, we are currently engaged in implementing a language called Orca (Bal and Tanenbaum, 1988).

Orca is based on the concept of globally shared objects. Programmers can define operations on shared objects, and the compiler and run time system take care of all the details of making sure they are carried out correctly. This scheme gives the programmer to ability to atomically read and write shared objects that are physically distributed among a collection of machines without having to deal with any of the complexity of the physical distribution. 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.

7.8. Performance

Performance, in general, has been a major success story. The minimum RPC time for Amoeba is 1.4 msec between two user-space processes on 16 MHz 68020s, and interprocess throughput is nearly 700 kilobytes per second. The file system lets us read and write files at the same rate.

7.9. User Interface

Amoeba 3.0 has a homebrew window system. In Amoeba 4.0, the X window system will replace the current window system which was designed before X existed. Although the current system is faster than X, many users will no doubt prefer X, since so much software exists for it and X is becoming something of a de facto standard.

7.10. Security

An intruder capable of tapping the network on which Amoeba runs can discover capabilities and do considerable damage. In a production environment some form of link encryption is needed to guarantee better security. Although some thought has been given to a security mechanism (Tanenbaum et al., 1986), it was not implemented in Amoeba 3.0.

Two security systems have been designed and implemented in Amoeba 4.0. The first version can only be used in fairly friendly environments where the network and operating system kernels can be assumed secure. This version uses one-way ciphers and, with caching of argument/result pairs, can be made to run virtually as fast as the current Amoeba. The other version makes no assumptions about the security of the underlying network or the operating system. Like MIT's Kerberos (Steiner et al., 1988) it uses a trusted authentication server for key establishement and encrypts all network traffic.

We intend to install both versions and investigate the effects on performance of the system. We are researching the problems of authentication in very large systems spanning multiple organizations and national boundaries.

8. CONCLUSION

The Amoeba project has clealy demonstrated that it is possible to build an efficient, high-performance distributed operating system on current hardware. The object-based nature of the system, and the use of capabilities provides a unifying theme that holds the various pieces together. By making the kernel as small as possible, 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.

Amoeba has been operating satisfactorily for several years now, both locally and to a limited extent over a wide-area network. Its design is clean and its performance is excellent. By and large we are satisfied with the results to date. Nevertheless, no operating system is ever finished, so we are continually working to improve it.

9. REFERENCES

Baalbergen, E.H. Design and Implementation of Parallel Make, *Computing Systems* 1., (Spring 1988), pp. 135-158.

Bal, H.E., Renesse, R. van, and Tanenbaum, A.S. Implementing Distributed Algorithms using

Remote Procedure Call, *Proc. National Computer Conference*, AFIPS, 1987. pp. 499-505.

Bal, H.E., and Tanenbaum, A.S. Distributed Programming with Shared Data, *IEEE Conf. on Computer Languages*, IEEE, 1988, pp. 82-91.

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

Burrows, M., and Schroeder, M.D. Performance of Firefly RPC, internal paper DEC SRC, Palo Alto, Cal., Nov. 1987.

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

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

Dennis, J., and Van Horn, E. Programming Semantics for Multiprogrammed Computation, *Commun. ACM* 9, (March 1966), pp. 143-155.

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.

Feldman, S.I. Make—A Program for Maintaining Computer Programs, *Software—Practice and Experience* 9, (April 1979) pp. 255-265.

Johnson, S.C. Yacc Yet Another Compiler Compiler, Bell Labs Technical Report, Bell Labs, Murray Hill, NJ, 1978.

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

Marsland, T.A., and Campbell, M. Parallel Search of Strongly Ordered Game Trees, *Computing Surveys* 14, (Dec. 1982) pp. 533-551.

Mullender, S.J., and Tanenbaum, A.S. The Design of a Capability-Based Distributed Operating System, *Computer Journal* 29, (Aug. 1986), pp. 289-299.

Mullender, S.J., and Tanenbaum, A.S. A Distributed File Service Based on Optimistic Concurrency Control, *Proc. Tenth Symp. Oper. Syst. Prin.*, 1985, pp. 51-62.

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, 1989a, pp. 22-27.

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

Renesse, R. van, Staveren, H. van, and Tanenbaum, A.S. Performance of the World's Fastest Distributed Operating System, *Operating Systems Review* 22, (Oct. 1988), pp. 25-34.

Renesse, R. van, and Tanenbaum, A.S. Voting with Ghosts, *Proc. Eighth Int'l Conf. on Distr. Comp. Systems*, IEEE, 1988, pp. 456-461.

Renesse, R. van, Tanenbaum, A.S., Staveren, H. van, and Hall, J. Connecting RPC-Based Distributed Systems Using Wide-Area Networks, *Proc. Seventh Int'l Conf. on Distr. Comp. Systems*, IEEE, 1987, pp. 28-34.

Steiner, J.G., Neuman, C., and Schiller, J.I. Kerberos An Authentication Service for Open Network Systems, *Proceedings of the Usenix Winter Conference*, USENIX Assoc., 1988, pp. 191-201.

Stonebraker, M. Operating System Su pport for Database Management, *Commun. ACM* 24, (July 1981), pp. 412-418.

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., Mullender, S.J., and Renesse, R., van Using Sparse Capabilities in a Distributed Operating System *Proc. Sixth International Conf. on Distr. Computer Systems*, IEEE, 1986.

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., and Renesse, R. van Distributed Operating Systems, *Computing Surveys* 17, (Dec. 1985) pp. 419-470.