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

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. Among the things done right were basing the system on objects, using a single uniform mechanism (capabilities) for naming and protecting them in a location independent way, and designing a completely new, and very fast file system. Among the things done wrong were having threads not be pre-emptable, initially building our own homebrew window system, and not having a multicast facility at the outset.

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

INTRODUCTION

The Amoeba project is a research effort aimed at understanding how to connect multiple computers together in a seamless way [15, 16, 26, 28, 32] The basic idea is to provide the users with the 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 4.0), 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 5.0.

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. A file's position in the directory hierarchy has no relation to its location.

The remainder of this paper will describe Amoeba and the lessons we have learned from building it. In the next section, we will give a technical overview of Amoeba as it currently stands. Since Amoeba uses the client-server model, we will then 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. Then we will discuss a number of applications that run on Amoeba. Measurements have shown Amoeba to be fast, so we will present some of our data. After that, we will discuss the successes and failures that we have encountered, so that others may profit from those ideas that have worked out well and avoid those that have not. Finally we conclude with a very brief comparision between Amoeba and other systems.

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.

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 also use X-terminals.

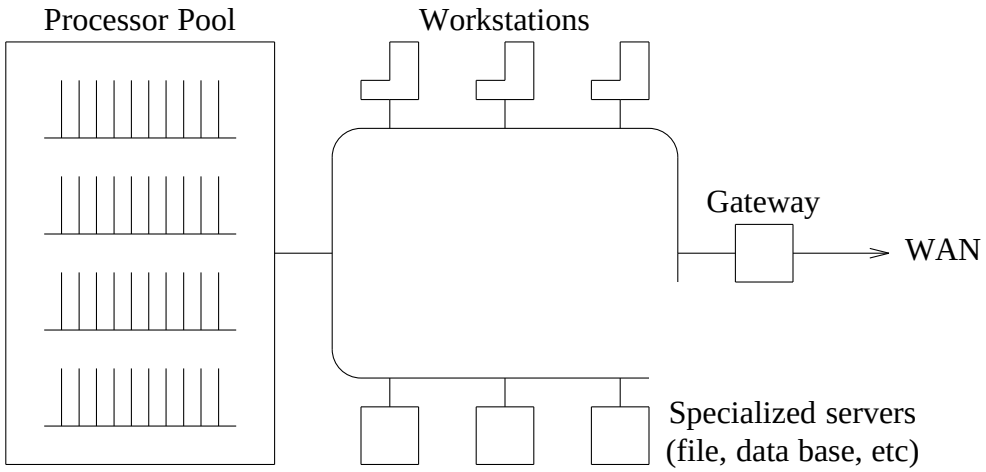

Fig. 1. The Amoeba architecture.

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, I/O, 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 (i.e., outside the kernel), providing for flexibility and experimentation.

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

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

This object-based model visible to the users is *implemented* using remote procedure call [5] 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 [27]).

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.

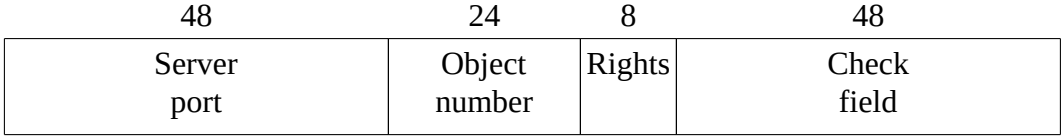

Fig. 2. A capability. The numbers are the current sizes in bits.

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. Since the operations are usually coarse grained, 8 bits is sufficient.

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 [9].

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.

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.

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 carefully designed and coded, lest they introduce unacceptable overhead.

To hide the marshalling and message passing from the users, Amoeba uses *stub routines* [5]. For example, one of the file system stubs might start with:

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

This call reads *nbytes* starting at *offset* from the file identified by *file_cap* into *buffer*. It returns the number of bytes actually read in *bytes_read*. The function itself returns 0 if it executed correctly or an error code otherwise. 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_hdr, req_buf, req_bytes, rep_hdr, buf, rep_bytes);
```

Automatic generation of such a stub from the procedure header above is impossible. Some essential information is missing. The author of the handwritten stub uses 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 [33]. The *read_file* operation could be part of an interface (called *class* in AIL) whose definition could look something like this:

```
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 client stub of the example above with the correct marshalling code. It can also generate the server main loop, containing the marshalling code corresponding to the client stubs. 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 must be a constant) 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 supports *multiple inheritance* so the Bullet server interface can inherit both the simple file interface and, for instance, a *capability management* interface for restricting rights on capabilities.

Currently, AIL generates stubs in C, but Modula stubs and stubs in other languages are planned. AIL stubs have been designed to deal with different data representations — such as byte order and floating-point representation — on client and server machines.

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 (optionally) synchronize with each other using mutexes or 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 thus effectively co-routines. 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 from 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. Our evaluation of this approach is discussed later.

SERVERS

The Amoeba kernel, as described above, essentially 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. Users who are not happy with the standard file system are free to write and use their 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 inappropriate it may be. In [24] for example, the numerous problems that UNIX creates for database systems are described at great length.

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

The Memory and Process 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. The key concepts are segments and process descriptors, as described below.

A *segment* is a contiguous chunk of memory that can contain code or data. Each segment has a capability that permits its holder to perform operations on it, such as reading and writing. A segment is somewhat like an in-core file, with similar properties.

A *process descriptor* is a data structure that provides information about a *stunned* process, that is, a process not yet started or one being debugged or migrated. It 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); 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.

A process is created by executing the following steps.

1. Get the process descriptor for the binary from the file system.
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*).
3. Modify the process descriptor to make the first segment the environment segment just created.
4. Send the process descriptor to the machine where it will be executed.

---

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

When the processor descriptor arrives at the machine where the process will run, the memory server there extracts the capabilities for the remote segments from it, and fetches the code and data segments from wherever they reside by using the capabilities to peform READ operations in the usual way. In this manner, the physical locations of all the machines involved are irrelevant.

Once all the segments have been filled in, the process can be constructed and the process started. A capability for the process is returned to the initiator. This capability can be used to kill the process, or it can be passed to a debugger to stun (suspend) it, read and write its memory, and so on.

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*) [17] was designed as an experiment in managing concurrent access using optimistic concurrency control. The current one, the *bullet server* was designed for extremely high performance [29, 31, 32]. 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 us to use a radically different design from that 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 below) more than offsets the small extra cost of having to buy, say, an 800 MB disk instead of a 500 MB disk in order to store 500 MB worth of files.

The bullet 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 (30,000 bytes), 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 straightforward to implement source code control systems.

The files are stored contiguously on disk, and are cached in the file server's memory (currently 12 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 high performance.

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 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. Earlier we discussed how the rights bits are protected from tampering by use of the check field.

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.

To see how how multiple columns are used, consider a typical access. The client provides a capability for a directory (implying a column) and a string. 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 for 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. The technique is similar to the one Eden used [20]. 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.

In addition to supporting replication of user objects, the directory server is itself duplicated. Among other properties, it is possible to install new versions of it by killing off one instance of it, installing a new version as the replacement, killing off the other (original) instance, and installing a second replacement also running the new code. In this way bugs can be repaired without interrupting service.

WIDE-AREA AMOEBA

Amoeba was designed with the idea that a collection of machines on a LAN 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 [30]. In particular, it is undesirable that the fast local RPC be slowed down 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 [7] it is expensive and inefficient. Furthermore, not every service should be available worldwide. For example, a laser printer server in 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 server 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

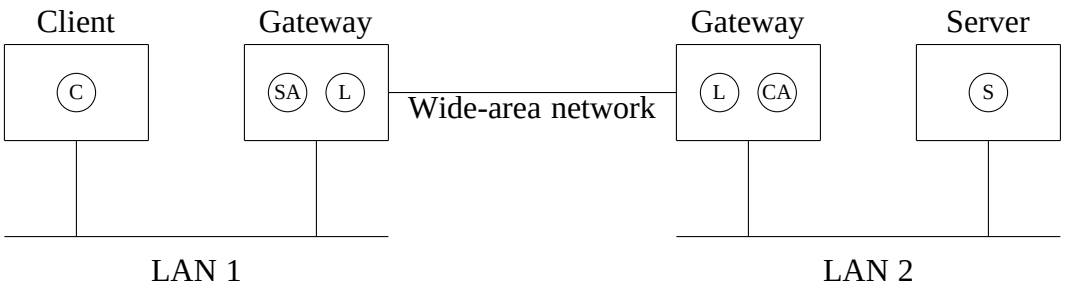

Fig. 4. Wide-area communication in Amoeba involves six processes.

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 normally 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 its management. 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.

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.

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 [25] compilers and other utilities on top of it.

Using a special set of library procedures that do RPCs with the Amoeba servers, it has been possible to construct an emulation of the UNIX system call interface — which was dubbed *Ajax* — 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. A *session server* has been provided to handle state information and do *fork* and *exec* in a UNIX-like way.

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* [10]. 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 being made but not yet fully generated.

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* [11] 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 [2].

The parallel compilations are directed by a new version of *make*, called *amake*. *Amake* does not use traditional *makefiles*. Instead, the user tells it which source files are needed, but not their dependencies. The compilers have been modified to keep track of the observed dependencies (e.g., which files they in fact included). After a compilation, this information goes into a kind of mini-database that replaces the traditional *makefile*. It also keeps track of which flags were used, which version of the compiler was used, and other information. Not having to even think about *makefiles*, not even automatically generated ones, has been popular with the users. The overhead due to managing the data base is negligible, but the speedup due to parallelization 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.

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 [13] and alpha-beta search [14] We briefly describe these below. More details can be found in [3].

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 return 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 75 percent of the theoretical maximum speedup can be achieved using this algorithm. The rest is lost to communication overhead.

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.

PERFORMANCE

Amoeba was designed to be 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 Sun 3/60s (20 MHz 68020s) using a 10 Mbps Ethernet. We measured the performance for three different configurations:

1. Two user processes running on Amoeba.
2. Two user processes running on Sun OS 4.0.3 but using the Amoeba primitives.
3. Two user processes running on Sun OS 4.0.3 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. The measurments represent the average values of 100,000 trials and are highly reproducible.

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. It should also be noted that the standard Amoeba header has room for 8 bytes of data, so in the test for 4 bytes, a only a header was sent and no data buffer. On the other hand, on the Sun, a special optimization is available for the local case, which we used.

In Fig. 5 we give the delay and the bandwidth of these eight cases, both for local processes (two distinct processes on the same machine) and remote processes (processes on 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 remote RPCs over the Ethernet.

	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	0.5	2.0	5.6	7.4	4000	5232
pure Amoeba remote	1.1	11.1	37.4	3.5	721	783
UNIX driver local	4.2	7.7	17.0	0.9	1039	1723
UNIX driver remote	5.1	26.7	88.6	0.8	300	331
Sun RPC local	5.7	12.8	imposs.	0.7	625	imposs.
Sun RPC remote	6.7	24.6	imposs.	0.6	325	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.

The interesting comparisons in these tables are the comparisons of pure Amoeba RPC and pure Sun OS RPC both for short communications, where delay is critical, and long ones, where bandwidth is the issue. A 4-byte Amoeba RPC takes 1.1 msec, vs. 6.7 msec for Sun RPC. Similarly, for 8 Kbyte RPCs, the Amoeba bandwidth is 721 Kbytes/sec, vs. only 325 Kbytes for the Sun RPC. The conclusion is that Amoeba's delay is 6 times better and its throughput is twice as good.

While the Sun is obviously not the only system of interest, its widespread use makes it a convenient benchmark. We have looked in the literature for performance figures from other

distributed systems and have shown the null-RPC latency and maximum throughput in Fig. 6.

System	*Hardware*	*Null RPC in msec.*	*Throughput in kbytes/s*	*Estimated CPU MIPS*	*Implementation Notes*
Amoeba	Sun 3/60	1.1	783	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. 6. Latency and throughput for some systems reported in the literature.

The RPC numbers for the other systems listed in Fig. 6. are taken from the following publications: Cedar [5], *x*-Kernel [19], Sprite [18], V [6], Topaz [22], and Mach [19].

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 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 [32] 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. 6 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 (Millions of Instructions Per Second). 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 peformance 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).

Similiarly, 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 [34].

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 transferred this way.

Topaz RPC was obtained on Fireflies, which are VAX-based multiprocessors. The performance obtained in Fig. 6 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 [19] 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).

Like Amoeba itself, the bullet server was designed with fast performance as a major objective. Below we present some measurements of what has been achieved. The measurements were made between a Sun 3/60 client talking to a remote Sun 3/60 file server equipped with a SCSI disk. Figure 7 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. In this case, the file is written to disk. Note that both the create and the delete operations involve disk requests,

Delay (msec)

File Size	READ	CREATE+DEL
1 Kbyte	2	50
16 Kbyte	20	84
1 Mbyte	1260	3210

(a)

Bandwidth (Kbytes/sec)

READ	CREATE+DEL
427	20
788	191
813	319

(b)

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

The careful reader may have noticed that a user process can pull 813 kbytes/sec from the bullet server (from Fig. 7), even though the user-to-user bandwidth is only 783 kbytes/sec (from Fig. 5). The reason for this apparent descrepancy is as follows. As far as the clients are concerned, the bullet server is just a black box. It accepts requests and gives replies. No user processes run on its machine. Under these circumstances, we decided to move the bullet server code into the kernel, since the users could not tell the difference anyway, and protection

is not an issue on a free-standing file server with only 1 process. Thus the 813 kbyte/sec figure is user-to-kernel for access to the file cache, whereas the 783 kbyte/sec one is user-to-user, from memory-to-memory without involving any files. The pure user-to-kernel bandwidth is certainly higher than 813 kbytes/sec, but some of it is lost to file server overhead.

To compare the Amoeba results with the Sun NFS file system, we have measured reading and creating files on a Sun 3/60 using a remote Sun 3/60 file server with a 16 Mbyte of memory running SunOS 4.0.3. The file server had the same type of disk as the bullet server, so the hardware configurations were, with the exception of extra memory for NFS, identical to those used to measure Amoeba. The measurements were made at night under a light load. To disable local caching on the Sun 3/60 we locked the file using the Sun UNIX *lockf* primitive while doing the read test. The timing of the read test consisted of repeated measurement of an *lseek* followed by a *read* system call. The write test consisted of consecutively executing *creat*, *write* and *close.* (The *creat* has the effect of deleting the previous version of the file.) The results are depicted in Fig. 8.

Delay (msec)

File Size	READ	CREATE
1 Kbyte	20	101
16 Kbyte	47	186
1 Mbyte	2030	13350

(a)

Bandwidth (Kbytes/sec)

READ	CREATE
50	10
340	86
504	77

(b)

Fig. 8. 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 results in lower bandwidths than for 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 capabilities, which gives it a relatively better performance for large files.

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 4.0 was found wanting, we will make improvements in Amoeba 5.0, 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 has also been implemented as part of MINIX [25] and as such is in widespread use around the world.

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 have no experience with capabilities on huge systems (thousands of simultaneous users). On one hand, with such a large system, some capabilities are bound to leak out, compromising security. On the other hand, capabilities provide a kind of firewall, since a compromised capability only affects the security of one object. Whether such fine-grained protection is better or worse in practice than more conventional schemes for huge systems is hard to say at this point.

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 5.0 will use 256-bit capabilites, rather than the 128-bit capabilities of Amoeba 4.0. The larger Check field will be more secure against attack. Other security aspects will also be tightened, including the addition of secure, encrypted communication between client and server. Also, the larger capabilities will 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 will be aligned at 32-bit boundaries, which potentially may give better performance.

Remote Procedure Call

For the most part, RPC communication is satisfactory, but sometimes it gives problems [27]. 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.

In Amoeba 4.0, when an RPC transfers a long message it is actually sent as a sequence of packets, each of which is individually acknowledged at the driver level (stop-and-wait protocol). Although this scheme is simple, it slows the system down. In Amoeba 5.0 we will only acknowledge whole messages, which will allow us to achieve higher bandwidths than shown

in Fig. 6.

Because RPC is inherently point-to-point, problems arise 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 finessed.

Amoeba 5.0 will fully support group communication using multicast. A message sent to a group will be delivered to all members, or at least an attempt will be made. A higher-level protocol has been devised to implement 100% reliable multicasting on unreliable networks at essentially the same price as RPC (two messages per reliable broadcast). This protocol is described in [12]. There are many applications (e.g., replicated data bases of various kinds) for which reliable broadcasting makes life much simpler. Amoeba 5.0 will use this replication facility to support fault tolerance.

Although not every LAN supports broadcasting and multicasting in hardware, when it is available (e.g., Ethernet), it can provide an enormous performance gain for many applications. For example, a simple way to update a replicated data base is to send a reliable multicast to all the machines holding copies of the data base. This idea is obvious and we should have realized it earlier and put it in from the start.

Although it has long since been corrected, in Amoeba 2.0 we made a truly dreadful decision to have asynchronous RPC. In that system the sender transmitted a message to the receiver and then continued executing. When the reply came in, the sender was interrupted. This scheme allowed considerable parallelism, but it was impossible to program correctly. Our advice to future designers is to avoid asynchronous messages like the plague.

Memory and Process Management

Probably the worst mistake in the design of the Amoeba 4.0 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 process 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.

The run-to-completion semantics of thread scheduling in Amoeba 4.0 also prevents a multiprocessor implementation from exploiting parallelism and shared memory by allocating different threads in one process to different processors. Amoeba 5.0 threads will be able to 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 semaphores or mutexes.

Another problem concerns the lack of timeouts on the duration of remote operations.

When the memory server is starting up a process, it uses the capabilities in the process descriptor to download the code and data. It is perfectly legal for these capabilities to be for somebody's private file server, rather than for the bullet server. However, if this server is malicious and simply does not respond at all, a thread in the memory server will just hang forever. We probably should have included service timeouts, although doing so would introduce race conditions.

Finally, Amoeba does not support virtual memory. It has been our working assumption that memory is getting so cheap that the added complexity of virtual memory is not worth it. Most workstations have at least 4M RAM these days, and will have 32M within a couple of years. Simplicity of design and implementation and high speed have always been our goals, so we really have not decided yet whether to implement virtual memory in Amoeba 5.0.

In a similar vein, we do not support process migration at present, even though the mechanisms needed for supporting it already exist. Whether process migration for load balancing is an essential feature or just another frill is still under discussion.

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 4.0 design, combining high performance with reliability, robustness, and ease of use.

An issue that we are becoming interested in is how one could handle databases in this environment. We envision an Amoeba-based database system that would have a very large memory for an essentially ''in-core'' database. Updates would be done in memory. The only function of the disk would be to make checkpoints periodically. In this way, the immutability of files would not pose any problems.

A problem that has not arisen yet, but might arise if Amoeba were scaled to thousands of users is caused by the splitting of the directory server and file server. Creating a file and then entering its capability into a directory are two separate operations. If the client should crash between them, the file exists but is inaccessible. Our current strategy is to have the directory server access each file it knows about once every k days, and have the bullet server automatically garbage collect all files not accessed by anyone in n days ($n >> k$). With our current setup and reliable hardware, this is not a problem, but in a huge, international Amoeba system it might become one.

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 4.0 is a system consisting of a large number of local area networks interconnected by many bridges and gateways. Although Amoeba 4.0 works on these systems, its performance is poor, partly due to the way port location and message handling is done. In Amoeba 5.0, 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 will be 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 the gateways for large systems.

One unexpected problem that we had was the poor quality of the wide-area networks that we had to use, especially the public X.25 ones. Also, to access some machines we often had to traverse multiple networks, each with their own problems and idiosyncracies. Our only insight to future researchers is not to blindly assume that public wide-area networks will actually function correctly until this has been experimentally verified.

UNIX Emulation

The Amoeba 4.0 UNIX emulation consists of a library and a session server. It was written 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 never be able to provide 100% compatibility. For example, the whole concept of user-ids and group-ids is very hard to get right in a capability-based system. Our view of protection is totally different.

Furthermore, Amoeba is essentially a stateless system. This means that various subtle properties of UNIX relating to how files are shared between parent and child are virtually impossible to get right. In practice we can live with this, but for someone who demanded binary compatibility, our approach has some shortcomings.

Parallel Applications

Although Amoeba was originally conceived as a system for *distributed* computing, the existence of the processor pool with 48 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 [4].

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 the 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 involving branch and bound, successive overrelaxation, and graph algorithms. For example, we have redone the traveling salesman problem in Orca and achieved a ten-fold speedup with 10 processors (compared to 7.5 using the non-Orca version described earlier). Alpha-beta search in Orca achieves a factor of 6 speedup with 10 processors (compared to 4 without Orca). It appears that using Orca reduces the communication overhead, but it remains true that for problems with many processes and a high interaction rate (i.e., small grain size), there will always be a problem.

Performance

Performance, in general, has been a major success story. The minimum RPC time for Amoeba is 1.3 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.

User Interface

Amoeba originally had a homebrew window system. It was faster than X-windows, and in our view cleaner. It was also much smaller and easier to understand. For these reasons we thought it would be easy to get people to accept it. We were wrong. Technical factors sometimes play second fiddle to political and marketing ones. We have abandoned our window server and switched to X windows.

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 [26] it was not implemented in Amoeba 4.0.

Two potential security systems have been designed for Amoeba 5.0. The first version can only be used in 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 [23] 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.

COMPARISON WITH OTHER SYSTEMS

Amoeba is not the only distributed system in the world. Other well-known ones include Mach [1], Chorus [21], V [6], and Sprite [18]. Although a comprehensive comparison of Amoeba with these would no doubt be very interesting, it is beyond the scope of this paper. Nevertheless, we would like to make a few general remarks.

The main goal of the Amoeba project differs somewhat from the goals of most of the other systems. It was our intention to develop a new operating system from scratch, using the best ideas currently available, without regard for backward compatibility with systems

designed 20 years ago. In particular, while we have written a library and server that provide enough UNIX compatibility that over 100 UNIX utilities run on Amoeba (after relinking with a special library), 100% compatibility has never been a goal. Although from a marketing standpoint, not aiming for complete compatibility with the latest version of UNIX may scare off potential customers with large existing software bases, from a research point of view, having the freedom to selectively use ideas from UNIX is a plus. Some other systems take a different viewpoint.

Another difference with other systems is our emphasis on Amoeba as a *distributed* system. It was intended from the start to run 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 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.

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. 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 (Tanenbaum), preferably by electronic mail.

REFERENCES

[1] 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.

[2] 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.

[3] Bal, H.E., Van Renesse, R., and Tanenbaum, A.S. Implementing Distributed Algorithms using Remote Procedure Call. *Proc. Nat. Comp. Conf.*, AFIPS, 1987. pp. 499-505.

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

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

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

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

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

[9] 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.

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

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

[12] 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.

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

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

[15] 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.

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

[17] Mullender, S.J., and Tanenbaum, A.S. A Distributed File Service Based on Optimistic Concurrency Control. *Proc. Tenth Symp. Operating System Principles*, (Dec. 1985), pp. 51-62.

[18] 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.

[19] 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.

[20] Pu, C., Noe, J.D., Proudfoot, A. Regeneration of Replicated Objects: A Technique and its

Eden Implementation. *Proc. 2nd Int'l Conf. on Data Eng.*, (Feb. 1986), pp. 175-187.

[21] 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.

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

[23] 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.

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

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

[26] Tanenbaum, A.S., Mullender, S.J., and Van Renesse, R. Using Sparse Capabilities in a Distributed Operating System. *Proc. Sixth International Conf. on Distr. Computer Systems*, IEEE, 1986.

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

[28] Tanenbaum, A.S., and Van Renesse, R. Distributed Operating Systems. *Computing Surveys* 17, (Dec. 1985), pp. 419-470.

[29] Van Renesse, R. 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.

[30] Van Renesse, R., Tanenbaum, A.S., Van Staveren, H., 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.

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

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

[33] Van Rossum, G. AIL—A Class-Oriented Stub Generator for Amoeba. *Proc. of the Workshop on Experience with Distributed Systems*, (J. Nehmer, ed.), Springer Verlag, 1990 (in preparation).

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