# The Performance Of The Amoeba Distributed Operating System

*Robbert van Renesse*
*Hans van Staveren*
*and*
*Andrew S. Tanenbaum*

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

*ABSTRACT*

Amoeba is a capability-based distributed operating system designed for high performance interactions between clients and servers using the well-known RPC model. The paper starts out by describing the architecture of the Amoeba system, which is typified by specialized components such as workstations, several services, a processor pool, and gateways that connect other Amoeba systems transparently over wide-area networks. Next the RPC interface is described. The paper presents performance measurements of the Amoeba RPC on unloaded and loaded systems. The time to perform the simplest RPC between two user processes has been measured to be 1.4 msec. Compared to SUN 3/50's RPC, Amoeba has 1/9 the delay, and over 3 times the throughput. Finally we describe the Amoeba file server. The Amoeba file server is so fast that it is limited by the communication bandwidth. To the best of our knowledge this is the fastest file server yet reported in the literature for this class of hardware.

## 1. INTRODUCTION

Many distributed operating systems have been designed [1]. Of the systems that have actually been built, only a few have grown beyond the stage of being a testbed for research into distributed applications to a generally usable distributed operating system. Often the reason is that the system is too slow to support real applications. This can be because the system is inherently slow, for example, because it has to provide a high degree of fault tolerance, or because it was built on top of another operating system, such as the UNIX† operating system, to facilitate development.

In this paper we describe the performance of the Amoeba distributed operating system [2, 3]. This system was designed to be used, and therefore we have devoted considerable energy to performance. The system uses the popular object-oriented model for distributed computing, in connection with remote procedure calls and lightweight processes.

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

† UNIX is a trademark of Bell Laboratories.

We report on the performance of the Amoeba interprocess communication, and of the file service. The measurements were performed on VME boards containing 16 MHz Motorola 68020 processors connected by a 10 Mbit Ethernet, and are compared to the performance of the commercially available SUN 3/50 UNIX system (using 15 MHz 68020s and SUN OS 3.5).

## 2. ARCHITECTURE OF THE AMOEBA SYSTEM

Amoeba is a distributed system being developed at the Vrije Universiteit and the Centre for Mathematics and Computer Science (CWI), both in Amsterdam. Amoeba currently runs on Motorola 68020, National Semiconductor 32032, and MicroVax II processors. Both Ethernet and the Pronet token ring are supported by Amoeba, and can be connected by a bridge.

The Amoeba architecture consists of four principal components, as shown in Fig. 1. First are the workstations, one per user, which run window management software, and on which users can carry out editing and other tasks that require fast interactive response [4, 5]. 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 [6, 7].

Third are the specialized servers, such as directory, file, and block servers, data-base servers, bank servers, boot servers, and various other servers with specialized functions. Fourth are the wide-area network gateways, which are used to link Amoeba systems at different sites in possibly different countries into a single, uniform system [8, 9]. 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 .start at 6/11 <L1.start, L1.end> line up 0.3i with .start at 7/11 <L1.start, L1.end> line up 0.3i 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 .start at 1/11 <L3.start, L3.end> line up 0.3i with .start at 2/11 <L3.start, L3.end> line up 0.3i 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 .start at 7/11 <L4.start, L4.end> line up 0.3i with .start at 8/11 <L4.start, L4.end> line up 0.3i 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 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

**Fig. 1.** The Amoeba architecture.

All the Amoeba machines run the same kernel, which primarily provides communication services and little else. The basic idea behind the kernel was to keep it small, not only to enhance its reliability, but also to allow as much of the operating system as possible to run as user processes, providing for flexibility and experimentation.

### 2.1. Transactions

Amoeba is an object-oriented distributed operating system. Objects are abstract data types such as files, directories, and processes, and are managed by server processes. A client process carries out operations on an object by sending a request message to the server process that manages the object. While the client blocks, the server performs the requested operation on the object. Afterwards the server sends a reply message back to the client, which unblocks the client. We have named this request/reply exchange a *transaction* (not to be confused with data-base transactions). Amoeba guarantees *at-most-once* execution of transactions. Remote procedure calls [10, 11] are implemented by assembling an operation code and its arguments in a request message, and performing a transaction with the appropriate server. The result of the procedure is retrieved from the reply message.

After starting a transaction, a client process blocks to await the reply. A server process blocks when it is awaiting a request. To handle multiple transactions going on at the same time a process can be subdivided into lightweight subprocesses called threads. By having a thread for each request, a server process can handle multiple requests simultaneously. A client process can perform several transactions at the same time by having a thread per transaction. To avoid race conditions and simplify programming, the threads are only rescheduled when the currently running thread blocks, that is, threads are non-pre-emptive.

### 2.2. Capabilities

All objects in Amoeba are named and protected by *capabilities*. [2, 12] Capabilities, combined with transactions, provide a uniform interface to all objects in the Amoeba system. A capability has 128 bits, and is composed of four fields, as shown in Fig. 2.

1) The *server port*: a 48 bit sparse address identifying the server process that manages the object. A server can choose its own port.
2) The *object number*: an internal 24 bit identifier that the server uses to distinguish among its objects. The server port and the object number together uniquely identify an object.
3) The *rights field*: 8 bits telling which operations on the object are permitted by the holder of this capability.
4) The *check field*: a 48-bit number that protects the capability against forging and tampering.

| 48 | 24 | 8 | 48 | # bits |
|---|---|---|---|---|
| Server Port | Object Number | Rights | Check Field | |

**Fig. 2.** A capability.

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 in the capability are protected by encrypting them together with the random number, and storing the result in the check field. A server can check a capability by performing the encryption operation again using the random number in the server's tables, and comparing the result with the check field in the capability.

Capabilities can be stored in directories that are managed by the *directory service*. A directory is effectively a set of <ASCII string, capability> pairs, and is itself just another object in the Amoeba system. Directory entries may, of course, contain capabilities for other directories, and thus an arbitrary naming graph can be built. The most common directory operation is to present an ASCII string and ask for the corresponding capability. Other operations are entering and deleting directory entries, and listing a directory.

## 3. THE AMOEBA INTERFACE

Request and reply messages in Amoeba consist of a header and a buffer. Headers are 32 bytes, and buffers can be up to 30,000 bytes. (In the near future this will be changed to 64 bytes and 1 gigabyte respectively.) 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 transaction primitives are listed in Fig. 3. To await a request message, a server calls GET-REQUEST specifying a header and a buffer in which to receive the request. A client invokes DO-TRANSACTION specifying the capability of the object to be operated on and the operation code in the request header. The server sends a reply using the PUT-REPLY primitive. Requests and replies are delivered reliably. Amoeba guarantees that messages are delivered at most once. The return status of DO-TRANSACTION can be one of three:

1) The request was delivered and has been executed. The size of the reply is returned.
2) The request was not delivered, and hence not executed (*e.g.*, a server could not be located).
3) The status is unknown: the request was sent, but any contact to the server was broken afterwards. The server may have crashed during the execution, leaving the state of the operating undefined. In this case the application level has to do its own fault recovery.

| |
|---|
| GET-REQUEST(req-header, req-buffer, req-size) |
| PUT-REPLY(rep-header, rep-buffer, rep-size) |
| DO-TRANSACTION(req-header, req-buffer, req-size, rep-header, rep-buffer, rep-size) |

**Fig. 3.** The Amoeba primitives.

It is important that the delay of small transactions be very low, and the bandwidth of large transactions be very high. The transaction interface is implemented by a small kernel that runs on every processor in the Amoeba system. A UNIX driver has been implemented that provides the same primitives to UNIX processes, allowing them to communicate with Amoeba clients and servers [13].

### 3.1. Implementation

A remote procedure call consists of more than just the request/reply exchange. 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. Moreover, it has to check the errors that might have occurred in the request/reply exchange. 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. We also have to handle different data representations in client and server. Also the capability checking might impose great overhead if not implemented carefully.

In the following sections we will briefly describe how the different parts in a remote procedure call have been implemented.

### 3.2. Protocol

When a client invokes DO-TRANSACTION for the first time, a packet containing the server port is broadcast over the network to request the physical location of the server [14]. The kernel running the server responds with a packet containing its physical network address. The client caches this information so that it may use it as a hint in subsequent transactions to the same server. Next the client sends the request packet, or a sequence of packets if the request does not fit in one packet, to the server using the acquired physical location. A retransmission timer is started to recover from network failures. Retransmissions are always sent to the same processor, since otherwise the at-most-once semantics cannot be guaranteed.

In the normal case in which a reply is generated quickly, the reply message is sent back and serves as the acknowledgement for the request. If the operation takes a long time, the client will retransmit the request. This time the server sends a separate acknowledgement. For a long transaction special packets are exchanged to enquire about the status of the transaction. Like requests, replies are split into several packets if they do not fit into one packet. Replies are separately acknowledged so that the server can start awaiting a new request immediately.

Special care needs to be taken to implement this protocol efficiently. First of all the coding has to be done carefully, since it turns out that the bottleneck in the communication is not the network, but in the processors that run the protocol. For example, unpacking densely packed messages are expensive operations. Second is the timer management. During a transaction many timers need to be started, but they hardly ever expire, since they are canceled when an expected packet arrives. An efficient way of implementing the timers is using a *sweep* algorithm, that periodically checks whether the protocol is still progressing. If not, a message might be lost and a retransmission is in order.

Third is the context switching. Often when a thread blocks there are no other threads to schedule, since there are many processors available in Amoeba and the work is balanced over the different processors. In this case it is unnecessary to remove the thread from the run queue. When a packet comes in for this thread it can be restarted from where it stopped, and there is no overhead in putting it back on the run queue again. Also, when the message consists of several packets, the protocol management can be done at the interrupt level, and the thread does not need to be restarted at all.

### 3.3. Marshalling

RPC requests usually consist of a number of integer parameters and sometimes a request buffer consisting of bytes. Replies usually consist of an integer result and sometimes a reply buffer consisting of bytes. Since this is the common case we have optimized its implementation. For example, read and write operations on a file usually consist of a buffer, an offset in the file, and a size. In the request and reply header we have reserved 8 bytes for parameters and results, which have been subdivided into two 2-byte words and a 4-byte word. These integer types have to be converted if the sender and receiver use different integer representations. The sender specifies which integer representation (little-endian or big-endian byte order) it uses.

More complicated data types can be handled by marshalling everything in the request and reply buffers. We leave the data representation in these buffers to the applications, but we have provided library routines that can be used to marshall common integer and floating point types in a machine-independent way. Work on a stub compiler is underway to have this done automatically.

The capability checking, if implemented naively, would involve expensive encryption for each operation. However, it is simple to cache the result of the encryption in the server, so that the encryption is hardly ever necessary. Cache entries are filled when capabilities are generated, or when the capability was not present in the cache. A simple leastrecently-used algorithm guarantees a high hit-rate.

### 3.4. Performance and Comparison

The performance measurements were performed on 16 MHz Motorola 68020 processors running the Amoeba kernel, and on SUN 3/50 workstations running SUN OS 3.5 UNIX. All processors were connected over the Ethernet using Lance chip interfaces (manufactured by Advanced Micro Devices). We have measured the performance for different configurations with clients and servers running on Amoeba, running under SUN UNIX using the Amoeba RPC driver, and running under SUN UNIX using the SUN RPC primitives. The load on the Ethernet not involved in the measurements can be ignored.

We will demonstrate the performance of the RPC mechanism using three common cases:

case 1)
**4 bytes**
In this test the request consists of, for example, a 4 byte integer, and there is an empty reply. Under Amoeba the 4 bytes will fit in the header, so both the request and reply are header only (no buffer).

case 2)
**8,192 bytes**
Under Amoeba the request is header only; the reply consists of a header plus an 8 Kbyte buffer. This could be, for example, a read operation of an 8 Kbyte file.

case 3)
**30,000 bytes**
The request is header only; the reply consists of a header plus a 30,000 bytes buffer. This is currently the maximum size of the Amoeba buffer. Since SUN RPC imposes a maximum message size of 8 Kbytes, this case could not be measured for SUN systems.

In Fig. 4. we give the delay and the bandwidth of the three different RPC examples for the different configurations. 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.

| | *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)* |
| bare Amoeba local | 0.8 | 2.5 | 7.1 | 5.0 | 3,277 | 4,255 |
| bare 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. 4.** The delay in msec (a) and bandwidth in Kbytes/sec (b) for 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. The UNIX driver implements Amoeba RPCs under SUN UNIX.

### 3.5. The Performance of Amoeba under Heavy Load

Amoeba RPC performs much better than SUN RPC on a lightly loaded network. For example, reading an 8K block from a remote file takes 13.1 ms between two Amoeba machines, and 40.6 ms between SUN 3/50 two machines running UNIX with SUN RPC. However, since Amoeba is a distributed operating system, RPCs under Amoeba are far more heavily used. It is therefore interesting to look at the behavior of Amoeba RPC under heavy load. In this section we will investigate two cases. The first case is where client/server pairs are trying to perform as many RPCs as possible on one network. In the second case, there is only one server, but several clients are doing as many RPCs as possible. The first case puts a heavy load on the network, and the second a heavy load on the server. In both cases there is one network, and one processor per client and per server.

There are two things that we want to measure. The first is how the performance of the Amoeba RPCs degrades with the number of clients. This should be no worse than just dividing the maximum performance over the clients. That is, if one client can do 700 RPCs per second, then two clients together should at least be able to do a total of 700 RPCs per second as well. We also want to know how fairly the RPCs are distributed over the clients. If, with two clients, one could execute only 5 RPCs, but the other did 695, then the scheduling of RPCs was unfair.

We have measured the performance and fairness of Amoeba RPC as a function of the number of clients, in each of the two cases. Each measurement is represented as shown in Fig. 5. The figure shows the average of the measurements, the minimum and maximum observed measurements, and a 95% confidence interval assuming normal (Gaussian) distribution of the measurements. The confidence interval is a measure of fairness. It gives the probability that a measurement falls within that interval. If the line representing the interval is short, the scheduling of the RPCs was fair. If the line is very short, it will be hidden behind the dot representing the average. A: line down 1i B: line right 0.2i from A.start - (0.1i, 0.2i) C: line right 0.2i from A.start - (0.1i, 0.8i) "●" at A.center - (0, 0.02i) "maximum " at B.center - (0.3i, 0) rjust "average " at A.center - (0.3i, 0) rjust "minimum " at C.center - (0.3i, 0) rjust arrow up 0.5i with .start at A.center + (0.4i, 0) " confidence" at last arrow.start + (0, 0.1i) ljust arrow down 0.5i with .start at A.center + (0.4i, 0) " interval" at last arrow.start - (0, 0.1i) ljust

**Fig. 5.** Measurements are represented by the average (●), the minimum and maximum observed measurements (—), and a 95% confidence interval ( | ).

Fig. 6 shows the results for pairs of clients and servers. Each client and server performs the same measurements as were done on a lightly loaded Ethernet. In Fig. 7(a) we see the result for null RPCs, that is, RPCs without any data. The dashed line gives the performance of one client/server pair divided by the number of clients, and represents graceful degradation. In this figure we can see that the pairs are not bothered by the load imposed by the other pairs. At least up to 5 client/server pairs, the performance as observed from each pair is about 700 RPCs per second. The fairness is about ideal.

dashwid=20; arrowwid=15; arrowht=20 A: [ arrow from (0, 0) to (0, 1100) "  # RPCs / second" at last arrow.end ljust arrow from (0, 0) to (1100, 0) "# clients" at last arrow.end - (0, 150) rjust line from (200, 0) to (200, -10) "1" at last line.end below line from (400, 0) to (400, -10) "2" at last line.end below line from (600, 0) to (600, -10) "3" at last line.end below line from (800, 0) to (800, -10) "4" at last line.end below line from (1000, 0) to (1000, -10) "5" at last line.end below line from (-10, 125) to (0, 125) "100  " at last line.start rjust line from (-10, 250) to (0, 250) "200  " at last line.start rjust line from (-10, 375) to (0, 375) "300  " at last line.start rjust line from (-10, 500) to (0, 500) "400  " at last line.start rjust line from (-10, 625) to (0, 625) "500  " at last line.start rjust line from (-10, 750) to (0, 750) "600  " at last line.start rjust line from (-10, 875) to (0, 875) "700  " at last line.start rjust line from (-10, 1000) to (0, 1000) "800  " at last line.start rjust "•" at (200, 884) line from (400, 889) to (400, 893) "•" at (400, 888) line from (393, 890) to (407, 890) line from (393, 892) to (407, 892) line from (600, 886) to (600, 902) "•" at (600, 891) line from (593, 890) to (607, 890) line from (593, 898) to (607, 898) line from (800, 883) to (800, 892) "•" at (800, 885) line from (793, 885) to (807, 885) line from (793, 890) to (807, 890) line from (1000, 807) to (1000, 844) "•" at (1000, 822) line from (993, 814) to (1007, 814) line from (993, 836) to (1007, 836) line from (200, 887) to (208, 849) line from (218, 812) to (228, 775) line from (240, 739) to (252, 702) line from (265, 667) to (280, 631) line from (296, 597) to (314, 563) line from (334, 530) to (355, 498) line from (378, 468) to (403, 439) line from (431, 411) to (459, 385) line from (490, 361) to (522, 339) line from (556, 318) to (591, 300) line from (626, 283) to (663, 267) line from (700, 253) to (737, 240) line from (775, 228) to (814, 217) line from (852, 208) to (891, 198) line from (930, 190) to (970, 182) ] "(a)" at A.s - (0, 0) below B: [ arrow from (0, 0) to (0, 1100) "  # RPCs / second" at last arrow.end ljust arrow from (0, 0) to (1100, 0) "# clients" at last arrow.end - (0, 150) rjust line from (200, 0) to (200, -10) "1" at last line.end below line from (400, 0) to (400, -10) "2" at last line.end below line from (600, 0) to (600, -10) "3" at last line.end below line from (800, 0) to (800, -10) "4" at last line.end below line from (1000, 0) to (1000, -10) "5" at last line.end below line from (-10, 90) to (0, 90) "2  " at last line.start rjust line from (-10, 181) to (0, 181) "4  " at last line.start rjust line from (-10, 272) to (0, 272) "6  " at last line.start rjust line from (-10, 363) to (0, 363) "8  " at last line.start rjust line from (-10, 454) to (0, 454) "10  " at last line.start rjust line from (-10, 545) to (0, 545) "12  " at last line.start rjust line from (-10, 636) to (0, 636) "14  " at last line.start rjust line from (-10, 727) to (0, 727) "16  " at last line.start rjust line from (-10, 818) to (0, 818) "18  " at last line.start rjust line from (-10, 909) to (0, 909) "20  " at last line.start rjust line from (-10, 1000) to (0, 1000) "22  " at last line.start rjust "•" at (200, 920) line from (400, 642) to (400, 685) "•" at (400, 661) line from (393, 656) to (407, 656) line from (393, 672) to (407, 672) line from (600, 419) to (600, 451) "•" at (600, 432) line from (593, 426) to (607, 426) line from (593, 440) to (607, 440) line from (800, 283) to (800, 448) "•" at (800, 362) line from (793, 332) to (807, 332) line from (793, 426) to (807, 426) line from (1000, 145) to (1000, 470) "•" at (1000, 304) line from (993, 226) to (1007, 226) line from (993, 418) to (1007, 418) line from (200, 923) to (208, 886) line from (217, 848) to (227, 811) line from (238, 775) to (250, 738) line from (262, 702) to (276, 666) line from (292, 631) to (309, 597) line from (327, 564) to (347, 531) line from (369, 500) to (393, 469) line from (418, 440) to (446, 413) line from (475, 388) to (506, 364) line from (539, 342) to (573, 322) line from (607, 303) to (643, 287) line from (680, 271) to (717, 257) line from (754, 244) to (793, 232) line from (831, 222) to (870, 212) line from (908, 203) to (947, 194) line from (987, 187) to (1026, 179) ] with .w at A.e + (500, 0) "(b)" at B.s - (0, 0) below

**Fig. 6.** The performance of Amoeba under load for (a) null RPCs, and (b) large RPCs of 30,000 bytes with five clients and five servers. The dashed line represents the performance of one client/server pair divided by number of clients, that is, linear degradation.

In Fig. 6(b) we see what happens for large RPCs (30,000 bytes). Remember that a single pair uses more than half the bandwidth of the Ethernet, so it is impossible that multiple pairs will not affect the performance of others in the measurements. However, together they put an even higher load on the Ethernet, such that the measurements are much better than expected from simple graceful degradation, as represented by the dashed line.

But we also observe that as the number of client/server pairs increases, the fairness decreases. With 5 clients, the minimum number observed was 5.0 RPCs per second (which is still 150,000 bytes per second), and the maximum number was 9.2 RPCs per second. Note that by now the load on the Ethernet is an unlikely 8 Mbits per second (80%),.

In Fig. 7 we find the measurements in the case where all clients are doing RPCs to the same server. Here we can observe that one client can nearly saturate the server with RPCs, since two clients do not put a much higher load on the server. The degradation is graceful, and in both the case of null RPCs (a) and large 30,000 byte RPCs the scheduling of RPCs in the server is reasonably fair. dashwid=20; arrowwid=15; arrowht=20 A: [ arrow from (0, 0) to (0, 1100) "  # RPCs / second" at last arrow.end ljust arrow from (0, 0) to (1100, 0) "# clients" at last arrow.end - (0, 150) rjust line from (100, 0) to (100, -10) "1" at last line.end below line from (200, 0) to (200, -10) "2" at last line.end below line from (300, 0) to (300, -10) "3" at last line.end below line from (400, 0) to (400, -10) "4" at last line.end below line from (500, 0) to (500, -10) "5" at last line.end below line from (600, 0) to (600, -10) "6" at last line.end below line from (700, 0) to (700, -10) "7" at last line.end below line from (800, 0) to (800, -10) "8" at last line.end below line from (900, 0) to (900, -10) "9" at last line.end below line from (1000, 0) to (1000, -10) "10" at last line.end below line from (-10, 125) to (0, 125) "100  " at last line.start rjust line from (-10, 250) to (0, 250) "200  " at last line.start rjust line from (-10, 375) to (0, 375) "300  " at last line.start rjust line from (-10, 500) to (0, 500) "400  " at last line.start rjust line from (-10, 625) to (0, 625) "500  " at last line.start rjust line from (-10, 750) to (0, 750) "600  " at last line.start rjust line from (-10, 875) to (0, 875) "700  " at last line.start rjust line from (-10, 1000) to (0, 1000) "800  " at last line.start rjust "•" at (100, 889) line from (200, 475) to (200, 475) "•" at (200, 472) line from (193, 475) to (207, 475) line from (193, 475) to (207, 475) line from (300, 303) to (300, 337) "•" at (300, 317) line from (293, 311) to (307, 311) line from (293, 328) to (307, 328) line from (400, 235) to (400, 244) "•" at (400, 237) line from (393, 236) to (407, 236) line from (393, 242) to (407, 242) line from (500, 185) to (500, 197) "•" at (500, 188) line from (493, 186) to (507, 186) line from (493, 194) to (507, 194) line from (600, 154) to (600, 166) "•" at (600, 157) line from (593, 155) to (607, 155) line from (593, 164) to (607, 164) line from (700, 131) to (700, 141) "•" at (700, 133) line from (693, 134) to (707, 134) line from (693, 141) to (707, 141) line from (800, 129) to (800, 143) "•" at (800, 133) line from (793, 132) to (807, 132) line from (793, 143) to (807, 143) line from (900, 91) to (900, 123) "•" at (900, 103) line from (893, 96) to (907, 96) line from (893, 115) to (907, 115) line from (1000, 86) to (1000, 103) "•" at (1000, 92) line from (993, 90) to (1007, 90) line from (993, 102) to (1007, 102) line from (100, 892) to (104, 854) line from (109, 816) to (114, 778) line from (120, 741) to (126, 703) line from (133, 666) to (141, 629) line from (150, 592) to (160, 556) line from (171, 520) to (184, 484) line from (198, 450) to (214, 416) line from (232, 383) to (253, 352) line from (276, 322) to (302, 294) line from (331, 269) to (362, 246) line from (395, 225) to (430, 207) line from (466, 191) to (503, 177) line from (541, 164) to (579, 153) line from (618, 144) to (657, 135) line from (696, 128) to (735, 121) line from (775, 115) to (814, 109) line from (854, 104) to (894, 99) line from (933, 95) to (973, 91) ] "(a)" at A.s - (0, 0) below B: [ arrow from (0, 0) to (0, 1100) "  # RPCs / second" at last arrow.end ljust arrow from (0, 0) to (1100, 0) "# clients" at last arrow.end - (0, 150) rjust line from (100, 0) to (100, -10) "1" at last line.end below line from (200, 0) to (200, -10) "2" at last line.end below line from (300, 0) to (300, -10) "3" at last line.end below line from (400, 0) to (400, -10) "4" at last line.end below line from (500, 0) to (500, -10) "5" at last line.end below line from (600, 0) to (600, -10) "6" at last line.end below line from (700, 0) to (700, -10) "7" at last line.end below line from (800, 0) to (800, -10) "8" at last line.end below line from (900, 0) to (900, -10) "9" at last line.end below line from (1000, 0) to (1000, -10) "10" at last line.end below line from (-10, 90) to (0, 90) "2  " at last line.start rjust line from (-10, 181) to (0, 181) "4  " at last line.start rjust line from (-10, 272) to (0, 272) "6  " at last line.start rjust line from (-10, 363) to (0, 363) "8  " at last line.start rjust line from (-10, 454) to (0, 454) "10  " at last line.start rjust line from (-10, 545) to (0, 545) "12  " at last line.start rjust line from (-10, 636) to (0, 636) "14  " at last line.start rjust line from (-10, 727) to (0, 727) "16  " at last line.start rjust line from (-10, 818) to (0, 818) "18  " at last line.start rjust line from (-10, 909) to (0, 909) "20  " at last line.start rjust line from (-10, 1000) to (0, 1000) "22  " at last line.start rjust "•" at (100, 921) line from (200, 491) to (200, 552) "•" at (200, 518) line from (193, 510) to (207, 510) line from (193, 532) to (207, 532) line from (300, 285) to (300, 411) "•" at (300, 345) line from (293, 311) to (307, 311) line from (293, 370) to (307, 370) line from (400, 215) to (400, 340) "•" at (400, 275) line from (393, 245) to (407, 245) line from (393, 315) to (407, 315) line from (500, 189) to (500, 250) "•" at (500, 217) line from (493, 207) to (507, 207) line from (493, 246) to (507, 246) line from (600, 142) to (600, 216) "•" at (600, 176) line from (593, 153) to (607, 153) line from (593, 199) to (607, 199) line from (700, 115) to (700, 189) "•" at (700, 149) line from (693, 125) to (707, 125) line from (693, 180) to (707, 180) line from (800, 81) to (800, 193) "•" at (800, 134) line from (793, 104) to (807, 104) line from (793, 180) to (807, 180) line from (900, 90) to (900, 155) "•" at (900, 119) line from (893, 96) to (907, 96) line from (893, 151) to (907, 151) line from (1000, 70) to (1000, 179) "•" at (1000, 122) line from (993, 90) to (1007, 90) line from (993, 156) to (1007, 156) line from (100, 924) to (104, 886) line from (108, 848) to (114, 810) line from (119, 772) to (125, 735) line from (132, 697) to (139, 660) line from (148, 623) to (157, 586) line from (167, 550) to (179, 514) line from (192, 479) to (207, 445) line from (224, 411) to (243, 379) line from (265, 348) to (289, 319) line from (316, 292) to (345, 267) line from (377, 244) to (410, 224) line from (446, 207) to (482, 191) line from (519, 177) to (557, 165) line from (595, 155) to (634, 145) line from (673, 137) to (712, 129) line from (751, 122) to (791, 116) line from (830, 111) to (870, 106) line from (910, 101) to (950, 97) line from (989, 93) to (1029, 89) ] with .w at A.e + (500, 0) "(b)" at B.s - (0, 0) below

**Fig. 7.** One server, many clients. (a) null RPCs,  (b) 30K RPCs.

## 4.  THE BULLET FILE SERVER

The price of memory is decreasing rapidly, allowing us to equip a file server with a large memory to radically improve its performance.  For Amoeba we have built such a fast file server, called the *Bullet server*.  This 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. An advantage of the immutability of files is that processes can cache them without having to worry about inconsistency. When an application wants to change the file, it reads the complete file into its memory. After making the required changes, a file is created in the Bullet server with the new contents. When the capability of the new file has been installed in the directory service, the new contents will be publicly available. This operation can be made atomic, even for a set of Bullet files, to achieve fault tolerance. Old files are automatically garbage collected.

### 4.1. Implementation

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.) Files are replicated on two disks. In the CREATE-FILE operation one can specify to reply before the file is written to disk, after it has been written to one disk, or after it has been written to both disks, depending on how important the stability of the file is.

Files are usually sent to the client as a whole, if possible in one large RPC reply. This way we are able to achieve the transfer rate that is provided by the RPC mechanism. The location of the file is kept in an ‘‘inode table,’’ containing the disk address, the size, and the random number of the file. The random number is used for capability checking. The inode table is kept contiguously at the beginning of the disk, and cached completely (write-through) in core.

### 4.2. Performance and Comparison

Figure 8 gives the performance of the Bullet file server for 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 both disks. 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 120 msec.

| | *Delay (msec)* | |
|---|---|---|
| File Size | READ | CREATE+DEL |
| 1 Kbyte | 3.0 | 130.0 |
| 16 Kbyte | 25.0 | 168.0 |
| 1 Mbyte | 1,550.0 | 4,160.0 |

(a)

| *Bandwidth (Kbytes/sec)* | |
|---|---|
| READ | CREATE+DEL |
| 341 | 7 |
| 650 | 98 |
| 677 | 252 |

(b)

**Fig. 8.** Performance of the Bullet file server for read operations, and create and delete operations together. The delay in msec (a) and bandwidth in Kbytes/sec (b) are given.

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. 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. 9.

| | *Delay (msec)* | | *Bandwidth (Kbytes/sec)* | |
|---|---|---|---|---|
| File Size | READ | CREATE | READ | CREATE |
| 1 Kbyte | 10.4 | 97.0 | 98 | 11 |
| 16 Kbyte | 47.0 | 191.0 | 349 | 86 |
| 1 Mbyte | 3,345.0 | 15,850.0 | 313 | 66 |
| | (a) | | (b) | |

**Fig. 9.** Performance of the SUN NFS file server for read and create operations. The delay in msec (a) and bandwidth in Kbytes/sec (b) are given.

Observe that reading and creating 1 Mbyte files result in lower bandwidths than reading and creating 16 Kbyte files. The Bullet file server performs for read operations 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 (120 msecs). For small files we therefore observe a lower bandwidth than for SUN NFS. Although the Bullet file server stores the files on two disks, for large files the bandwidth is four times that of SUN NFS.

## 5. CONCLUSIONS

We have discussed the design and implementation of Amoeba. Amoeba is based on the object model and uses remote procedure calls to operate on objects. To make it a usable system, considerable effort has been devoted to providing high performance. This has been achieved by simple, yet carefully designed and implemented RPC protocols. Security has not been ignored in this process.

Two important aspects of RPC performance are delay and bandwidth. Compared to the SUN RPC, Amoeba executes a small RPC 9 times faster, and achieves over 3 times the bandwidth for large RPCs. Amoeba also performs well under high load, providing its users with a fair share of the available bandwidth.

We have also measured the performance of the file service of Amoeba, called the Bullet file server. While providing high availability through replication, the file service also provides high performance. Again by simple, but careful design and implementation, the delay and bandwidth of reading files are as good as the Amoeba RPC. Considering the high capability-based file protection, and the fact that files are replicated on two disks, the write performance is also excellent. Compared to SUN NFS, the Amoeba file server is over twice as fast for reading large files, and four times faster for writing large files. The measurements convince us that a fast distributed operating system can be built.

## 6. ACKNOWLEDGEMENTS

We would like to thank Henri Bal, Greg Sharp, Jennifer Steiner, and the referees, for their critical reading of the manuscript and their valuable suggestions.

## 7. REFERENCES

[1] Tanenbaum, A. S. and Renesse, R. van, ‘‘Distributed Operating Systems,’’ *ACM Computing Surveys*, Vol. 17, No. 4, pp. 419-470, December 1985.

[2] Mullender, S. J. and Tanenbaum, A. S., ‘‘The Design of a Capability-Based Distributed Operating System,’’ *The Computer Journal: ,* Vol. 29, No. 4, pp. 289-300, March 1986.

[3] Mullender, S. J. and Tanenbaum, A. S., ‘‘Protection and Resource Control in Distribut ed Operating Systems,’’ *Computer Networks* , Vol. 8, No. 5-6, pp. 421-432, Oct. 1984.

[4] Sharp, G. J., ‘‘The Design of a Window System for Amoeba,’’ Report IR-142, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, December 1987.

[5] Renesse, R. van, Tanenbaum, A. S., and Sharp, G. J., ‘‘The Workstation: Computing Resource or Just a Terminal?,’’ *Proc. of the Workshop on Workstation Operating Systems*, Cambridge, MA, November 1987.

[6] Bal, H. E., Renesse, R. van, and Tanenbaum, A. S., ‘‘Implementing Distributed Algorithms Using Remote Procedure Calls,’’ *Proc. of the 1987 National Computer Conf* ., pp. 499-506, Chicago, IL, June 1987.

[7] Baalbergen, E. H., ‘‘Parallel and Distr. Compilations in Loosely-Coupled Systems: A Case Study,’’ *Proc. Workshop on Large Grain Parallelism*, Prov., RI, Oct. 1986.

[8] Renesse, R. van, Tanenbaum, A. S., Staveren, J. M. van, and Hall, J., ‘‘Connecting RPC-Based Distributed Systems Using Wide-Area Networks,’’ *Proc. of the 7th Int. Conf. on Distr. Computing Systems*, pp. 28-34, West Berlin, September 1987.

[9] Renesse, R. van, et. al., ‘‘MANDIS/Amoeba: A Widely Dispersed Object-Oriented Operating System,’’ *Proc. of the EUTECO 88 Conf.*, pp. 823- 831, ed. R. Speth, North-Holland, Vienna, Austria, April 1988.

[10] Birrell, A. D. and Nelson, B. J., ‘‘Implementing Remote Procedure Calls,’’ ,I "ACM Trans. Comp. Syst." , Vol. 2, No. 1, pp. 39-59, February 1984.

[11] Spector, A. Z., ‘‘Performing Remote Operations Efficiently on a Local Computer Network,’’ *Comm. ACM*, Vol. 25, No. 4, pp. 246-260, April 1982.

[12] Tanenbaum, A. S., Mullender, S. J., and Renesse, R. van, ‘‘Using Sparse Capabilities in a Distributed Operating System,’’ *Proc. of the 6th Int. Conf. on Distr. Computing Systems*, pp. 558-563, Cambridge, MA, May 1986.

[13] Renesse, R. van, ‘‘From UNIX to a Usable Distributed Operating System,’’ *Proc. of the EUUG Autumn ’86 Conf.*, pp. 15-21, Manchester, UK, September 1986.