The Evolution of a Distributed Operating System

Robbert van Renesse
Andrew S. Tanenbaum

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam

Sape J. Mullender

Centre for Mathematics and Computer Science
Amsterdam

AMOEBA is a research project to build a true distributed operating system using the object model. Under the COST11-ter MANDIS project this work was extended to cover wide-area networks. Besides describing the system, this paper discusses the successive versions in the implementation of its model, and why the changes were made. Its purpose is to prevent ourselves and others from making the same mistakes again, and to illustrate how a distributed operating system grows in sophistication and size.

1. Why This Paper

"Those who learn nothing from history are doomed to repeat it" — Santayana

For about eight years now, we have been doing research on distributed operating systems, not only behind our desks, but also behind our terminals. The distributed system we are developing is called AMOEBA[1], and it is constantly evolving. It is 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.

COST11-ter MANDIS is an international project investigating the management requirements for large international networks of computers. It has adopted the object-model as a framework within which to discuss the management of wide-area distributed systems. To experiment with this, the MANDIS project adopted the Amoeba distributed operating system, extended with a gateway for wide-area communication. Amoeba systems in Holland (Vrije Universiteit, CWI), the U.K. (Harwell Laboratories, Hatfield Polytechnic), in Berlin (GMD/FOKUS) and in Norway (University of Tromsø) have been connected into a single, transparent distributed system.

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

In any system, mistakes can appear in the design: features that are missing, features that are obsolete, and features that are too hard to handle. Sometimes the solution needs a considerable redesign of the system, and a new version is born. One has to be prepared to redo systems [2-4]. When designing a system, it is important not to make mistakes twice, be they your own, or anyone else's. Therefore it is necessary to read about other comparable projects, and to document your own.

2. The AMOEBA Architecture

Bradley's Bromide: "If computers get too powerful, we can organize them into a committee—that will do them in"

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 [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, $5 \times 6 = 30$ processors could be allocated for the six compilations, gaining even more speedup [6].

Third are the specialized servers, such as directory [7], file servers [8], 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, such as investigated in the MANDIS work [9-13].

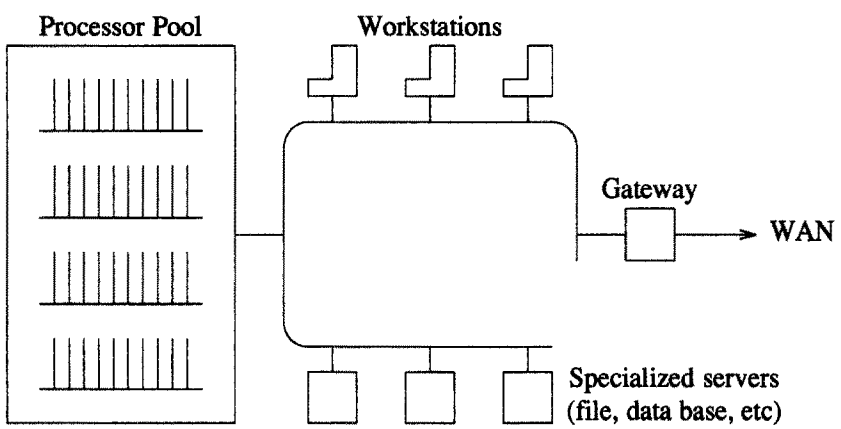

Fig. 1. The AMOEBA architecture.

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, 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) [14, 15]. AMOEBA guarantees *at-most-once* execution of transactions. Remote procedure calls [16, 17] are implemented by collecting a code identifying the procedure to be executed and the 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 not pre-empted.

2.2. Capabilities

All objects in AMOEBA are named and protected by *capabilities* [18, 19]. 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:

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 tell which of its objects this is. 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.

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 as well as a 48-bit random number. The index into the table is used as the object number in 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 capability can be checked by performing the encryption operation again, 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 [7].

3. AMOEBA Incarnations

"Experience is that marvelous thing that enables you to recognize a mistake when you make it again" — F.P. Jones

To get experience with distributed operating systems, and with the object model in particular, we have built an implementation of the AMOEBA system. This implementation consists of a small, highly portable, and efficient kernel, capable of providing local and remote communication, driving peripherals, and running processes; all other services are provided by user processes. In the following we only discuss the kernel.

Working with the first version of AMOEBA, we became aware of some of the deficiencies in its design. After a while we threw it away and built a new version. As this version did not solve all the problems, we designed and implemented the third, and current, version. We are currently designing the fourth version. Each of these versions are discussed more or less independently in the next sections. In section 4 we will compare them and describe why the changes were made.

3.1. AMOEBA 1.0

The AMOEBA 1.0 kernel [20] is a simple multiprogramming kernel, with intra-machine communication based on software interrupts. It has three layers. The bottom layer catches all hardware interrupts. Each interrupt causes a message to be put into a *task queue*. Messages may contain parameters, such as the value of a character just received on a communication line. Mostly these are the values of some of the devices that generated the interrupt. Furthermore, the layer schedules the kernel *tasks*, that constitute the middle layer of the kernel, and the user processes in the highest layer.

A task takes care of a particular device, for example, a disk or a clock. It is called whenever there is a message for it on the task queue. A user process is scheduled when there are no tasks left to run, or if the current running process has eaten up its time slice. Both tasks and processes are able to put something in the task queue, thus scheduling a task.

Tasks run to completion. When an interrupt occurs, a message is put on the task queue, and the task is resumed. This means that there are no race conditions in interrupt handling, and only one run-time stack is needed for all tasks. Tasks can be programmed entirely in a high-level language.

The two most important tasks are the clock task and the network task. The clock task simulates multiple timers: it has functions to set and cancel timers. The network task provides a network interface that does not guarantee delivery. A user process needs both services to implement a reliable network interface.

A user process can suspend itself, enable or disable certain messages from specific sources, and send or cancel messages. It invokes a task by sending a message to it by placing an entry in the task queue. The message contains four parameters, such as the specific function that must be executed by the task. As in the kernel, these messages are queued when arriving inconveniently. When a process is properly enabled, it is informed that a message is pending by an interrupt.

This way a process can call the three functions performed by the network task: *get(header, buffer)*, *put(header, buffer)*, and *unget(header)*. The header (see Fig. 2) is a 40

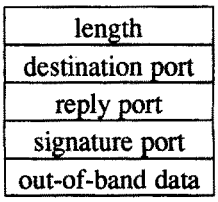

Fig. 2. Header format.

byte structure containing the total length of header and buffer, the destination *port*, the reply *port*, the *signature port*, and 20 byte out of band data. A port is a network independent address, chosen from a sparse 48 bit address space, and protected by a cryptographic one-way function. The signature port can be used for sender authentication.

Get enables receiving, while *put* sends a packet. Neither are 100% reliable in that packets may get lost. An interrupt is generated on completion. *Unget* disables receiving. The data buffer has a maximum size of 2 Kbytes, enough to contain about two thirds of the files in an average file system.

A user library of procedures uses these primitives, together with timers, to implement the transaction interface. A client invokes a service by calling *trans(hdr1, buf1, hdr2, buf2)*. The request is put into *hdr1* and *buf1*; the reply will be put into *hdr2* and *buf2*. The server calls *getreq(header, buffer)* to enable receiving of a request, and *putrep(header, buffer)* to send a reply back. In each of the three calls, an interrupt is generated on completion.

The protocol used is simple, yet makes efficient use of the network bandwidth. Normally the reply acknowledges the request, and the reply is acknowledged by the next request. Separate acknowledgements are generated only when the reply or the next request is taking too long. It is possible to have multiple outstanding *getreq*'s, to handle more than one client, or to have multiple *trans*'s going on, thus enabling parallel programming.

3.2. AMOEBA 2.0

Intra-machine communication in AMOEBA 2.0 is through 26-byte typed messages, called *mini-messages*. When a hardware interrupts occurs, the real-time information is put into a mini-message and sent to the appropriate task. The user interface to the tasks is also through these messages. This kernel has formed the base for the MINIX operating system [21].

The calls to send and receive messages are:

send(destination, message);
recv(source, message);
sendrec(destination, message).

Send sends the message to the specified destination: tasks are identified by negative numbers, processes by positive numbers. When the destination is not ready to receive, the message is queued. *Recv* is called when a task or process wants to await a message from the specified source, which may be ANY. *Sendrec* is provided for efficiency: it sends the message to the destination and awaits a reply message.

A process may be interrupted by a task or another process with a special interrupt message. Interrupts go at most one level deep, to simplify interrupt handling. Other messages that arrive during interrupt handling are queued as usual.

The services provided by the kernel are the same as in AMOEBA 1.0., including the clock task and the network task. The transaction mechanism interface to the user processes is almost identical, so existing user services for AMOEBA 1.0 are easily ported. Later the transaction interface of AMOEBA 3.0, the currently used incarnation of AMOEBA, has been implemented for MINIX.

3.3. AMOEBA 3.0

In AMOEBA 3.0, all communication, both intra-machine and inter-machine, is through transactions. The interface is slightly modified and extended:

getreq(header, buffer, length);
putrep(header, buffer, length);
trans(hdr1, buf1, len1, hdr2, buf2, len2);
new_thread(procedure);
thread_exit();
sleep(event);
wakeup(event).

The server, either a kernel task or a user process, calls *getreq* to await a request message, and *putrep* to send a reply back. A client process calls *trans* to send the request in *hdr1* and *buf1*, and to await the reply, which will be put into *hdr2* and *buf2*. The header contains the capability identifying the service and object, and 20 bytes of out of band data containing the command to the server and its parameters. The buffer, with a specified *length* of maximally 30 Kbytes, contains the data associated with the request or the reply.

Note that these calls are blocking, and prevent parallel computing. To allow concurrent programming, we introduce *threads*, a light-weight sub-process. Within a process, only one thread can run at a time; another one may be scheduled when the current running thread does a blocking call. While some threads are awaiting a request or a reply, another thread may run. A server that wants to be able to service multiple clients will have several identical threads, created with *new_thread*, capable of executing requests.

The kernel is just another process, having threads (tasks) to drive the peripherals. The bottom layer in the kernel schedules the threads of all processes, executes the transactions, copies local messages, and runs the network protocol. Device interrupts are still queued, but not transformed into messages. Instead, interrupt routines are invoked at "save" times, that is, in between thread switches. The network protocol sends separate acknowledgements for request and reply fragments, and network DMA is done simultaneously with the other side as much as possible. No separate timers are maintained, but a simple, once in a while "sweep" procedure restarts stopped protocols. All this results in simple and efficient message passing [14, 15].

The physical location of ports, and thus of servers and objects, is maintained in a cache per site. When the location of a port is unknown or out-of-date, it is located with a special broadcast *locate message*, and the cache is updated.

Threads within a process can synchronize using *sleep* and *wakeup*. A thread that wants to await an event invokes *sleep*; a thread that wants to resume other threads waiting for a

certain event calls *wakeup*. Since threads run to the next blocking system call, there is no danger of race conditions.

Under the COST11-ter project AMOEBA 3.0 was extended to wide-area networks using a special gateway [11, 12]. The gateway manages the wide-area communication without affecting the local networks. This management includes naming and protection of objects, accounting, and fault management of communication. The gateway is high-level: it intercepts complete messages, and if access is granted, establishes a virtual circuit to the intended destination to forward the message across. The gateway at the destination site repeats the transaction, and forwards the reply over the same virtual circuit back to the source The gateway registers all remote servers and their locations to know which messages to forward and which not. site.

3.4. AMOEBA 4.0

In AMOEBA 4.0 [22] processes are subdivided into light-weight threads, but now we no longer guarantee that threads run unpreempted to the next blocking system call. Moreover, we allow threads to await requests for multiple ports, and to specify message buffers of up to one Gigabyte. This has affected the user interface as follows:

getreq(port-list, header, buffer, length);
putrep(header, buffer, length);
trans(hdr1, buf1, len1, hdr2, buf2, len2);
new_thread(procedure);
thread_exit();
mu_lock(mutex);
mu_trylock(mutex, timeout);
mu_unlock(mutex).

Note that *sleep* and *wakeup* cannot be used as synchronization primitives anymore, since they would be fraught with race conditions because of the preemptive scheduling of tasks. *mu_lock* and *mu_unlock* respectively acquire and release a mutex variable. *mu_trylock* tries to acquire the lock within *timeout* milliseconds, and returns an error if this fails.

An important change in this new incarnation of AMOEBA is the format of the capability, which, as we will see, also influences the semantics of *trans*. The new format is shown in Fig. 3. The sizes of the different fields have been increased. Moreover, there is an extra field designating the creation site. In AMOEBA 4.0 it is assumed that objects hardly ever migrate away from the site of their creation. This obsoletes the necessity to register all remote services at the gateways, thus decreasing the amount of management necessary considerably.

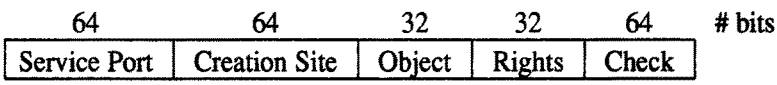

Fig. 3. An AMOEBA 4.0 capability.

4. Comparison

"I have made mistakes but I have never made the mistake of claiming that I never made one" — James Gordon Bennett

Having discussed each of the implementations of the object model more or less independently, it is now time to look what changed and why. The differences concern efficiency and programmability; these goals are often conflicting. Both metamorphoses are discussed in the following sections.

4.1. AMOEBA 1.0 → AMOEBA 2.0

Our first objection against AMOEBA 1.0 was the difficulty in programming with it. All communication with the outside world was through asynchronous messages. Although flexible, it puts us back some decades when programmers had to work at a very low level. The processes were pelted with interrupts. Each process had to do its own job scheduling.

Furthermore, the interrupts carried too little information—often additional information had to be transported by a special copying task. In addition to the complexity involved, it was inefficient, and it had protection problems. When data had to be copied between two processes, one had to do this, and thus had full access to the address space of the other process.

Furthermore, debugging was difficult, because it was hard to trace a process that can be interrupted at any moment, and each time somewhere else. Moreover, interrupts might arrive in another order when the process was executed again, and debug statements in the code changed the behaviour. Thus AMOEBA 1.0 processes were nondeterministic, and a failure might occur only once in a month, making it hard to find the error.

We abolished these problems in AMOEBA 2.0 by abolishing needless interrupts. All ordinary communication was through typed mini-messages, and although small, they were large enough for an average command with parameters or a reply. Messages only arrived when called for, which made both programming and debugging considerably easier, because a program could be written in the usual structured way.

4.2. AMOEBA 2.0 → AMOEBA 3.0

Although happier, we were not completely satisfied with our basis for a distributed operating system. To begin with, too little concurrency was left in with the new intra-machine communication mechanism. The receive call was blocking, and it was not possible to check if there was something in the message queue. Moreover, it was not possible to give a set of sources from which to receive a message, so the messages had to be handled in the order they arrived.

Also annoying were the different intra-machine and inter-machine communication mechanisms. This problem also existed in AMOEBA 1.0, but in AMOEBA 2.0 the mechanisms are much more alike. Furthermore, to start a transaction, a mini-message had to be sent to the network task, another to enable receipt of the acknowledgement, and a third to the clock task to set a timer. When the acknowledgement arrived, the timer had to be canceled, which cost another mini-message. All this made inter-machine communication inefficient.

These problems were solved in AMOEBA 3.0 by making the transaction the only communication primitive. Moreover, the messages are much larger, so a special task to copy data

became obsolete. At the same time, the protection problem with copying disappeared. Communication became transparent, having obvious advantages.

Like the mini-message calls, transaction calls were blocking now. Concurrent programming was made possible through threads: each thread can handle one client and one server. This way we have the profit of concurrent programming combined with the ease of simple, every-day programming.

4.3. AMOEBA 3.0 → AMOEBA 4.0

AMOEBA 3.0 is the first incarnation that is heavily used for distributed applications [23-25], and has led to several suggestions for improvements. Also, the hardware technology has improved considerably, making multi-processors more and more interesting. In the first three incarnations we envisioned only loosely-coupled hardware, but now we also have to deal with processors sharing memory over a shared bus. Yet another factor that makes a new implementation necessary is the advance of wide-area networks, making large distributed operating systems interesting.

There are two reasons for preemptive scheduling of threads. The first reason is one of software engineering. Due to the high level of transparency, the programmer cannot be expected to know if the standard library routine for printing makes calls to a remote printer or not. It was bad programming practice to rely on procedures being local, and thus trusting that no scheduling would occur. Therefore the advantages of non-preemptive scheduling largely disappeared. The other reason for preemptive scheduling of threads is that the performance of a multi-threaded process can be increased by running the different threads on different processors in a multi-processor.

The other important change in AMOEBA 4.0 is the Creation Site field in capabilities. This has to do with scaling. It was found unfeasible to have a purely flat name space that would cover the world [26, 27]. Using the old capability, it was impossible to transparently locate the server for the object in a world-wide AMOEBA system. Now, with the new capability lay-out, requests for operations on an object can be sent to the site that created the object immediately, where the server can then be located using the old broadcast-oriented mechanisms. In the rare event that an object migrates between AMOEBA sites, a *forwarding server* has to be left behind at the site that created the object to forward the request to the site where the object actually lives.

5. What We Have Learned

"The only thing we learn from history is that we learn nothing from history" — Hegel

The versions we have implemented, and the reasons for making them, have now been discussed. It is time to look why we went wrong in the design and to learn our lessons, to prove Hegel was wrong.

In the design of AMOEBA 1.0 we aimed at a simple and efficient kernel, and forgot the user interface. We did not appreciate the importance of the simplicity and the functionality of the user interface enough, which is an error in any system. Furthermore, in implementing the inter-machine interface, we forgot that its efficiency was likewise important.

In the design of AMOEBA 2.0 we were determined not to make the same mistakes again, so we concentrated too much on have a clean user interface, and did not worry about efficiency. The interface was not flexible enough, and too much intra-machine communication

was necessary to send a simple message, because the decomposition into layers and modules was too finely grained.

In AMOEBA 3.0 the networking primitives were made an integral part of the operating system instead of a separate attached task. This made all communication transparent and resulted in a high performance [14, 15]. Under the COST11-ter MANDIS work a gateway was added that made international communication transparent.

The last incarnation, AMOEBA 4.0, was developed mainly to deal with new technologies of multi-processors and wide-area networks. Using the experience gained with AMOEBA 3.0, several small changes where made to the system.

We feel that we are converging to a good distributed operating system. This paper shows the importance of implementing prototype systems for the development of a large distributed operating system. Prototype systems produce the flaws in the design of the system and give the necessary experience for developing the next version. It is necessary to document the mistakes to avoid making them again.

6. References

[1] 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).

[2] Lampson, B. W., "Hints for Computer System Design," *Proc. of the 9th ACM Symp. on Operating Systems Principles*, New York (October 1983).

[3] Tanenbaum, A. S. and Renesse, R. van, "Making Distributed Systems Palatable," *Proc. of 2nd SIGOPS Workshop Making Distr. Systems Work*, Amsterdam (September 1986).

[4] Mullender, S. J., "Making Amoeba Work," *Proc. of 2nd SIGOPS Workshop Making Distr. Systems Work*, Amsterdam (September 1986).

[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] Baalbergen, E. H., "Design and Implementation of Parallel Make," *Computing Systems*, Vol. 1, No. 2, pp. 135-158 (Spring 1988).

[7] Renesse, R. van and Tanenbaum, A. S., "A Directory Service supporting Availability and Consistency," *internal report* (1989).

[8] Renesse, R. van, Tanenbaum, A. S., and Wilschut, A., "The Design of a High-Performance File Server," *Proc. of the 9th Int. Conf. on Distr. Computing Systems*, Newport Beach, CA (June 1989).

[9] Langsford, A. E. and others, "Distributed Systems in Wide-Area Networks," pp. 96-104, in *Proc. European Telematics Conf.*, Elsevier Science Pub., Amsterdam (October 1983).

[10] Hall, J., Renesse, R. van, and Staveren, J. M. van, "Gateways and Management in an Internet Environment," *Proc. of the IFIP TC6 WG6.4A Int. Workshop on LAN Management*, Hahn-Meitner-Institute, Berlin (West) (July 1987).

[11] 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, Berlin (West) (September 1987).

[12] Renesse, R. van, Staveren, J. M. van, Hall, J., Turnbull, M., Janssen, A. A., Jansen, A. J., Mullender, S. J., Holden, D. B., Bastable, A., Fallmyr, T., Johansen, D., Mullender, K. S., and Zimmer, W., "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).

[13] Bacon, J. M., Horn, C., Langsford, A., Mullender, S. J., and Zimmer, W., "MANDIS: Architectural Basis for Management," *Proc. of the EUTECO 88 Conf.*, pp. 795-809, ed. R. Speth, North-Holland, Vienna, Austria (April 1988).

[14] Renesse, R. van, Staveren, J. M. van, and Tanenbaum, A. S., "The Performance of the World's Fastest Distributed Operating System," *ACM Operating Systems Review*, Vol. 22, No. 4, pp. 25-34 (October 1988).

[15] Renesse, R. van, Staveren, J. M. van, and Tanenbaum, A. S., "The Performance of the Amoeba Distributed Operating System," *Software—Practice and Experience*, Vol. 19, No. 3, pp. 223-234 (March 1989).

[16] Birrell, A. D. and Nelson, B. J., "Implementing Remote Procedure Calls," *ACM Trans. Comp. Syst.*, Vol. 2, No. 1, pp. 39-59 (February 1984).

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

[18] Mullender, S. J. and Tanenbaum, A. S., "Protection and Resource Control in Distributed Operating Systems," *Computer Networks*, Vol. 8, No. 5-6, pp. 421-432 (October 1984).

[19] 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).

[20] Tanenbaum, A. S. and Mullender, S. J., "A Simple, Efficient Multiprogramming Kernel," Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam (1982).

[21] Tanenbaum, A. S., "Operating Systems—Design and Implementation," Prentice-Hall, Englewood Cliffs, NJ (1987).

[22] Mullender, S. J., Jansen, A. J., and Rossum, G. van, "Amoeba Kernel Interface Specification," Centre for Mathematics and Computer Science, Amsterdam (March 1988).

[23] 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, Ill (June 1987).

[24] Bal, H. E. and Renesse, R. van, "A Summary of Parallel Alpha-Beta Search Results," *ICCA Journal*, Vol. 9, pp. 146-149 (September 1986).

[25] Johansen, D. and Anshus, O. J., "A Distributed Diary Application," *Proc. of the IFIP TC 6 First Iberian Conf. on Data Communications*, ed. A. Cerveira., North-Holland, Lisbon, Portugal (May 1987).

[26] Mullender, S. J. and Vitányi, P. M. B., "Distributed Match-Making for Processes in Computer Networks," *Proc. of the 4th ACM Conf. on Principles of Distr. Computing*, Minaki, Canada (August 1985).

[27] Mullender, S. J. and Vitányi, P. M. B., "Distributed Match-Making," *Algorithmica, 2nd special issue on distributed algorithms* (1988).