# Using Sparse Capabilities in a Distributed Operating System

*Andrew S. Tanenbaum*

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

*Sape J. Mullender*

Centre for Mathematics and Computer Science
Amsterdam, The Netherlands

*Robbert van Renesse*

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands
Most distributed operating systems constructed to date have lacked a unifying mechanism for naming and protection.
In this paper we discuss a system, Amoeba, that uses capabilities for naming and protecting objects.
In contrast to traditional, centralized operating systems, in which capabilities are managed by the operating system kernel, in Amoeba all the capabilities are managed directly by user code.
To prevent tampering, the capabilities are protected cryptographically.
The paper describes a variety of the issues involved, and gives four different ways of dealing with the access rights.

## 1. INTRODUCTION

Capabilities [Dennis and Van Horn 1966] have been used as the basis for a variety of uniprocessor operating systems (see [Levy 1984] for numerous examples). They have the attraction of providing a single, uniform mechanism for naming, accessing, and protecting all objects within the system. In all of these systems, the capabilities are managed by (trusted) kernel software, often with special assistance from the hardware.

The use of capabilities as a conceptual base for distributed systems has been minimal to date, a few exceptions being the Eden system [Almes et al. 1985], LINCS [Donnelley 1981], and ACCENT [Rashid 1981], Our scheme also uses a distributed capability mechanism, but it differs from each of these in significant ways, which we will describe after discussing our proposal.

This paper describes a scheme in which user processes manipulate capabilities directly in their own address spaces. Except for some very special parts of it, the kernel does not even know that capabilities are in use. To prevent users from forging new capabilities or tampering with existing ones, capabilities are protected cryptographically. This cryptographic protection scheme will first be described in some detail, followed by a discussion of how these capabilities are used in the Amoeba distributed operating system.

## 2. PORTS AND CAPABILITIES

### 2.1. Background on Amoeba

Amoeba is an object-oriented distributed operating system. Its semantic model is based on having client processes perform operations on objects managed by server processes. Objects are specified by capabilities. Operations are carried out by having processes exchange messages, generally in the form of a request from a client followed later by a reply from a server. The standard message format provides a place for one capability in the header, typically for the object being operated on, but users are free to put other capabilities in the data field as required. The header also contains room for the operation code and some parameters.

After making a request, a client blocks until the reply comes in, so the approach can be regarded as a simple remote procedure call mechanism [Spector 1982; Birrell and Nelson 1984]. The system does not use "connections" or virtual circuits or any other long-lived communication structures.

### 2.2. Ports

Every server has one or more *ports* to which client processes can send messages to contact the service (i.e., the server process). Ports consist of large numbers, typically 48 bits, which are known only to the server processes that comprise the service, and to the server's clients. For a public service, such as the file system, the port will generally be made known to all users. The ports used by an ordinary user process will, in general, be kept secret. Knowledge of a port is taken by the system as prima facie evidence that the sender has a right to communicate with the service. Of course the service is not required to carry out work for clients just because they know the port, for example, the file server will refuse to read or write files for clients lacking appropriate file capabilities. Thus two levels of protection are used here: ports for protecting access to servers, and capabilities for protecting access to individual objects. These two mechanisms are related, as will be shown later.

Although the port mechanism provides a convenient way to provide partial authentication of clients ("if you know the port, you may at least talk to the service"), it does not deal with the authentication of servers. How does one insure that malicious users do not listen on the file server's port, and try to impersonate the file server to the rest of the system?

One approach is to have all ports manipulated by kernels that are presumed trustworthy and are supposed to know who may listen on which port. As mentioned above, we reject this strategy because on some machines, e.g., personal computers users may be able to tamper with the operating system kernel, and also because we believe that by making the kernel as small as possible, we can enhance the reliability of the system as a whole. Instead, we have chosen a different solution that can be implemented in either hardware or software.

In the hardware solution, we need to place a small interface box, which we call an F-box (Function-box) between each processor module and the network. The most logical place to put it is on the VLSI chip that is used to interface to the network. Alternatively, it can be put on a small printed circuit board inside the wall socket through which personal computers attach to the network. In those cases where the processors have user mode and kernel mode and the operating systems can be trusted, it could be put into operating system. In any event, we assume that somehow or other all messages entering and leaving every processor undergo a simple transformation that users cannot bypass.

The transformation works like this. Each port is really a pair of ports, *P*, and *G*, related by: $P = F(G)$, where F is a (publicly-known) one-way function [Wilkes 1968; Purdy 1974; Evans et al. 1974] performed by the F-box. The one-way function has the property that given *G* it is a straightforward computation to find *P*, but that given *P*, finding *G* is not feasible.

Using the one-way F-box, the server authentication can be handled in a simple way, as illustrated in Fig 1. Each server chooses a get-port, *G*, and computes the corresponding put-port, *P*. The get-port is kept secret; the put-port is distributed to potential clients or in the case of public servers, is published. When the server is ready to accept client requests, it does a GET(*G*). The F-box then computes $P = F(G)$ and waits for messages containing *P* to arrive. When one arrives, it is given to the process that did GET(*G*). To send a message to the server, the client merely does PUT(*P*), which sends a message containing *P* in a header field to the server. The F-box on the sender's side does not perform any transformation on the *P* field of the outgoing message. boxwid=1i boxht=0.5i A: box "Client" B: box with .sw at A.sw + (4,0) "Server" C: box with .sw at A.ne + (1i, 1i) "Intruder" boxwid=0.25i boxht=0.25i D: box at A.n + (0, 0.25i) "F" E: box at B.n + (0, 0.25i) "F" F: box at C.s - (0, 0.25i) "F" line from A.n to D.s line from D.n to D.n + (0, 0.125i) line from C.s to F.n line from F.s to F.s - (0, 0.125i) line from B.n to E.s line from E.n to E.n + (0, 0.125i) line from A.nw + (-0.5i, 0.5i) to B.ne + (0.5i, 0.5i)

**Fig. 1.** Clients, servers, intruders and F-boxes.

Now let us consider the system from an intruder's point of view. To impersonate a server, the intruder must do GET(*G*). However, *G* is a well-kept secret, and is never transmitted on the network. Since we have assumed that *G* cannot be deduced from *P* (the one-way property of F) and that the intruder cannot circumvent the F-box, he cannot intercept messages not intended for him. An intruder doing GET(*P*) will simply cause his F-box to listen to the (useless) port F(*P*). Replies from the server to the client are protected the same way, only with the client picking a get-port for the reply, say, $G'$, and including $P' = F(G')$ in the request message.

The presence of the F-box makes it easy to implement digital signatures for still further authentication, if that is desired. To do so, each client chooses a random signature, *S*, and publishes $F(S)$. The F-box must be designed to work as follows. Each message presented to the F-box for transmission contains three special header fields: destination (*P*), reply ($G'$), and signature (*S*). The F-box applies the one-way function to the second and third of these, transmitting the three ports as: *P*, $F(G')$, and $F(S)$, respectively. The first is used by the receiver's F-box to admit only messages for which the corresponding GET has been done, the second is used as the put-port for the reply, and the third can be used to authenticate the sender, since only the true owner of the signature will know what number to put in the third field to insure that the publicly-known $F(S)$ comes out.

It is important to note that the F-box arrangement merely provides a simple *mechanism* for implementing security and protection, but gives operating system designers considerable latitude for choosing various *policies*. The mechanism is sufficiently flexible and general that it should be possible to put it into hardware without precluding many as-yet-unthought-of operating systems to be designed in the future. In effect, it is a protected associative addressing scheme. The associative addressing can be simulated in software when the kernels are trusted by having each one maintain a cache of (port, machine-number) pairs. If a port is not in the cache, it can be found by broadcasting a LOCATE message. How this can be carried out efficiently, even in a network without broadcasting, is discussed in [Mullender and Vitanyi 1984], along with many of the implications of location dependendent addressing, process migration, etc.

### 2.3. Capabilities

In any object-based system, a mechanism is needed to keep track of which processes may access which objects and in what way. The normal way is to associate a capability with each object, with bits in the capability indicating which operations the holder of the capability may perform. In a distributed system this mechanism should itself be distributed, that is, not centralized in a single monolithic "capability manager." In our proposed scheme, each object is managed by some server, which itself is a user (as opposed to kernel) process, and which understands the capabilities for its objects.

A capability typically consists of four fields as illustrated in Fig. 2.

1. The put-port of the server that manages the object
2. An object number meaningful only to the server managing the object
3. A rights field, containing a 1 bit for each permitted operation
4. A random number, for protecting each object

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

**Fig. 2.** A capability.

The basic model of how capabilities are used and protected can be illustrated by a simple example: a client wishes to create a file using the file server, write some data into the file, and then give another client permission to read (but not modify) the file just written. To start with, the client sends a message to the file server's put-port specifying that a file is to be created. The request might contain a file name, account capability, etc. The server would then pick a random number, store this number in its object table, and insert it into the newly-formed object capability. The reply would contain this capability for the newly created (empty) file.

To write the file, the client would send a succession of data messages, each containing the capability and some data. When each WRITE request arrived at the file server process, the server would use the OBJECT field contained in the capability as as index into its file tables to locate the object. For a UNIX† like file server, the object number would be the i-number, which could be used to locate the i-node.

Several object protection systems are possible using this framework. In the simplest one, the server merely compares the random number in the file table (put there by the

† UNIX is a trademark of Bell Laboratories.

server when the object was created) to the one contained in the capability. If they agree, the capability is assumed to be genuine, and all operations on the file are allowed. This system is easy to implement, but does not distinguish between READ, WRITE, DELETE, and other operations that may be performed on objects.

However, the basic idea can easily be modified to provide that distinction. We will now describe three different algorithms for protecting the access rights. In the first version, when a file (object) is created, the random number chosen and stored in the file table is used as an encryption/decryption key. The capability is built up by taking the RIGHTS field, which is initially all 1s to indicate that all operations are legal, and the RANDOM field (e.g., 48 bits), which contains a known constant, say, 0, and treating them as a single number. This number is then encrypted by the key just stored in the file table, and the result put into the newly minted capability in the combined RIGHTS-RANDOM field.

When the capability is returned for use, the server uses the OBJECT field (not encrypted) to find the file table and hence the encryption/decryption key. If the result of decrypting the capability leads to the known constant in the RANDOM field, the capability is almost assuredly valid, and the RIGHTS field can be believed. Clearly, an encryption function that mixes the bits thoroughly is required to ensure that tampering with the Rights Field also affects the known constant. EXCLUSIVE-OR'ing a constant with the concatenated RIGHTS and RANDOM fields will not do.

A second algorithm for protecting the RIGHTS field makes use of one-way functions, similar to the way ports are protected. When a server is asked to create a new object, it generates a random number, as usual. The RIGHTS field is then EXCLUSIVE-ORed with the random number and then used as the argument of the one-way function, *F*, yielding a value that is put into the RANDOM field of the capability. Symbolically,

RANDOM field = *F*(random-number XOR rights bits) .

The RIGHTS field is included in the capability itself in plaintext. When a capability arrives at the server, it finds the original random number from its internal tables and EXCLUSIVE-OR's the plaintext RIGHTS field with it, passing this result through *F*. If the result agrees with the RANDOM field in the capability, the capability is considered valid. Although a user can tamper with the plaintext RIGHTS field, such tampering will result in the server ultimately rejecting the capability.

When either of these protection systems are used, the owner of an object can easily give an exact copy of its capability to another process by just sending it the bit pattern, but to pass, say, read-only access, is slightly harder. To accomplish this task, the process must send the capability back to the server along with a bit mask and a request to fabricate a new capability with fewer rights.

This idea works well except that it requires going back to the server every time a sub-capability with fewer rights is needed. We will now describe a third algorithm that does not have this drawback. To start with, find a set of *N* commutative one-way functions, $F_0, F_1, \cdots, F_{N-1}$ corresponding to the N rights present in the RIGHTS field. When an object is created, the server chooses a random number and puts it in both its internal table and the RANDOM field, just as in the very first scheme presented. The server also sets all the RIGHTS field bits to 1.

A client can delete permission $k$ from a capability by replacing the RANDOM field, R, with $F_k(R)$ and turning off the corresponding bit in the RIGHTS field. When a capability comes into the server to be used, the server fetches the original random number from its table, looks at the RIGHTS field and applies the functions corresponding to the deleted rights to it. If the result agrees with the number present in the capability, then the capability is accepted as genuine, otherwise it is rejected.

Note that although the RIGHTS field is not encrypted, it is pointless for a client to tamper with it, since the server will detect that. In theory at least, the RIGHTS field is not even needed, since the server could try all $2^N$ combinations of the functions to see if any worked. Its presence merely speeds up the checking. It should also be clear why the functions must be commutative—it does not matter in what order the bits in the RIGHTS field were turned off. This scheme is discussed in more detail in [Mullender 1985].

The organization of capabilities and objects discussed above has the interesting property that although no central record is kept of who has which capabilities, it is easy to revoke existing capabilities. All that the owner of an object need do is ask the server to change the random number stored in its internal table and return a new capability. Obviously this operation must be protected with a bit in the RIGHTS field, but if it succeeds, all existing capabilities for that object are instantly invalidated.

### 2.4. Protection without F-Boxes

Earlier we said that protection could also be achieved in software (i.e., without F-boxes). It is slightly more complicated, since it uses both conventional and public-key encryption [Diffie and Hellman 1976], but it is still quite usable. The basic idea underlying the method is the fact that in nearly all networks an intruder can forge nearly all parts of a message being sent except the source address, which is supplied by the network interface hardware. To take advantage of this property, imagine a (possibly symmetric) conceptual matrix, $M$, of conventional (e.g., DES) encryption keys, with the rows being labeled by source machine and the columns by destination machine. Thus the matrix selects a unique key for encrypting the *capabilities* in any message. The data need not be encrypted, although that is also possible if needed.

Each machine is assumed to know the contents of its row and column of the matrix, and nothing else (how this will be achieved will be discussed shortly). Thus a client $C$ will know $M_{CX}$ and $M_{XC}$ for all $X$, and a server $S$ will know $M_{SX}$ and $M_{XS}$, all of which are conventional (not public) keys. With this arrangement, intruder $I$ can easily capture messages from client $C$ to server $S$, but attempts to "play them back" to the server will fail because the server will see the source machine as $I$ (assumed unforgeable) and use element $M_{IS}$ as the decryption key instead of the correct $M_{CS}$. No matter what the intruder does, he cannot trick the server into using a decryption key that decrypts the capabilities to make sense.

To avoid having to run the encryption/decryption algorithm frequently, all machines can maintain a hashed cache of capabilities that they have been using frequently. Clients will hash their caches on the unencrypted capabilities in the form of triples: (unencrypted capability, destination, encrypted capability), whereas servers will hash theirs in the form of triples: (encrypted capability, source, unencrypted capability).

To set up the matrix initially, the following procedure can be used. A public server, such as a file server, makes its put-port and a public encryption key known to the whole world. When a new machine joins the network (e.g., after a crash or upon initial system boot), it sends a broadcast message announcing its presence. Suppose, for example, the file server has just come up, and must (1) prove that it is the file server to other processes, and (2) establish the conventional keys used for encrypting capabilities in both directions.

A client machine, *C*, which receives the broadcast from the alleged file server, *F*, picks a new conventional encryption key, *K*, for use in subsequent *C* to *F* traffic and sends it to *F* encrypted with *F*'s public key. *F* then decrypts *K* and replies to *C* by sending a message containing both *K* and a newly chosen conventional key to be used for reverse traffic. This message is encrypted both with *K* itself and with the inverse of *F*'s public key, so *C* can use *K* and *F*'s public key to decrypt it. If the decrypted message contains *K*, *C* can be sure that the other conventional key was indeed generated by owner of *F*'s public key, thus convincing *C* that he is indeed talking to the file server. Both of the above-mentioned conditions have now been fulfilled, so normal communication can now take place. Note that the use of different conventional keys after each reboot make it impossible for an intruder to fool anyone by playing back old messages.

Yet another any possibility for protecting capabilities in the absence of F-boxes is to use conventional link-level encryption on all the data communication lines.

## 3. USE OF CAPABILITIES IN AMOEBA

In the preceeding sections we have seen how capabilities can be cryptographically protected so that they can be managed directly by user processes throughout the distributed system, without any help, or even knowledge by the operating system kernels. In the following sections we will look at some of the areas these capabilities have been applied in the Amoeba distributed operating system. The areas to be covered are: the memory server, the block server, the flat file server, the directory server, the multiversion file server, and the bank server. Capabilities are also used in other areas, but space limitations prevent them from being discussed here.

### 3.1. The Memory Server

The memory server is a process that manages physical memory and processes at the lowest level. It is actually part of the kernel present on each machine, but it communicates with other processes via the normal message protocol so that its clients do not perceive it as being special in any way.

The memory server is typically used for creating processes, as follows. The parent process tells the memory server to CREATE SEGMENT, providing an initial size and some other information. The memory server then returns a capability for the newly created segment. Using this capability, the parent process can use the WRITE operation to load data into the segment (the READ operation can get it back again later if needed). The parent process will normally repeat this cycle, creating and loading segments until all the child process' initial segments have been constructed, for example, text, data, and stack segments.

To create the child process, the parent then performs a MAKE PROCESS operation, providing the capabilities for the child's segments as parameters. The memory server then returns a process capability for the child, with which the child can be started, stopped, and generally manipulated. By directing the CREATE SEGMENT requests to a memory server on a remote machine, the parent can create the child wherever it wants to, providing a more convenient and efficient interface than the traditional FORK + EXEC.

The memory server can also easily support an "electronic disk." An electronic disk of the required size is created using CREATE SEGMENT, and then can be read and written, either by local or remote processes using READ and WRITE.

### 3.2. The Block Server

The Amoeba file system also makes heavy use of capabilities. As far as the operating system is concerned, a file system is just one or more server processes, with no special privileges. This design makes it possible to have multiple, potentially quite different file systems running at the same time. Three distinct file systems have in fact been implemented.

The first file system is highly modular, consisting of a block server, flat file server, and directory server. The block server can be requested to allocate a disk block and return a capability for it. Using this capability, the block can be written, read, or deallocated. The block server has no concept of a file. By splitting the block server off from the file server, it becomes possible for any user to implement any kind of special-purpose file system that he needs, without having to get into the details of disk storage management.

### 3.3. The Flat File Server

The flat file server provides its clients with files consisting of a linear sequence of bytes, numbered from 0 to the file size - 1. The basic operations here are CREATE FILE, DESTROY FILE, WRITE FILE, and READ FILE. CREATE FILE returns a capability used in the other calls, each of which implicitly specifies a file via the capability, and a position in the file via a parameter. The server does not have any concept of an "open" file. One can operate on any file for which a valid capability can be presented.

### 3.4. The Directory Server

The directory server manages directories, each of which is a set of (ASCII name, capability) pairs. A typical operation is to present the directory server with the capability for a directory, plus an ASCII string, and ask it to look up and return the capability that corresponds to the given string in the given directory. Operations also exist to enter and remove (ASCII name, capability) entries from directories. These primitives, and a few others, provide an adequate basis for building up aribtrary directory trees, graphs, etc. Note that the capabilities within a directory need not all be file capabilities and certainly need not all be located in the same place or managed by the same server. To look up the path a/b/c relative to some directory, a client would ask the server to find the string "a" in that directory. If the capability returned happens to be for a directory managed by a different directory server, then the ensuing request to look up "b" just goes to the new server. Unless the client compared the SERVER fields in the two capabilities, it wouldn't even notice that succeeding requests were going to different servers. The distribution is completely transparent.

### 3.5. The Multiversion File Server

The second file system supports tree-shaped files. Each file consist of a tree of pages, rather than a simple linear byte sequence. An important property of this file system is its ability to provide atomic updates on files. In short, a user can ask to make a new version of a file, which results in a capability for the new version. The new version acts like it is a page-by-page copy of the original, although in fact, pages are only copied when they are changed.

The new version can be modified at will, and then atomically "committed," thus becoming the new file. A file is thus a sequence of versions. Once a version of a file has been committed, it cannot be modified. This technique has been designed for use with video disks and other "write once" media. More details can be found in [Mullender and Tanenbaum 1982].

The third file system is a capability-based UNIX file system, to ease the problem of moving existing applications from UNIX to Amoeba.

### 3.6. The Bank Server

Resource control and accounting also makes use of the capabilities. The basis for the resource control and accounting is the bank server, which manages "bank account" objects. The principal operation on bank accounts is transferring virtual money from one account to another. Thus to obtain permission to create a file, a client would present a capability for one of his accounts to the bank server, and request that the bank server withdraw some money from that account and deposit it in the account of the file server. Assuming the client trusts the file server, the client can pre-pay for a substantial amount of work, in order to eliminate the overhead of going back to the bank on each request.

The bank server is prepared to maintain accounts in different, possibly convertible, possibly inconvertible, currencies. This mechanism can form the basis of a variety of policies, used by different servers. For example, by having the file server charge *x* dollars per kiloblock of disk space, quotas can be implemented by limiting how many dollars each client has. CPU time could be charged in francs, phototypesetter pages in yen, and so on. In some cases (e.g., disk blocks, but not typesetter pages), returning the resource might result in the client getting his money back

## 4. DISCUSSION

In this paper we have shown how ports and capabilities can be managed in a protected way in a distributed operating system. By moving the entire capability management out of the kernel, we can provide a minimal kernel, and yet have a powerful and general conceptual basis for naming and protection throughout the system. A number of examples of how capabilities are used in Amoeba were presented as examples.

The Eden [Almes et al. 1985] and ACCENT [Rashid 1981] systems also use capability-like mechanisms for protection, butin both cases, the ultimately responsiblity for managing the capabilities rests with the kernel. In Eden, users may manage capabilities directly, but the kernel maintains copies, to be able to verify each one before it is used. We maintain that moving all of the capability management out of the kernel is a step in the right direction. Just as file servers are now rarely part of the kernel of distributed systems, capability management should not be either. The smaller and simpler the kernel, the easier it is to write, debug, and maintain. Furthermore, if the system consists of a building full of rooms with wall sockets into which any user can plug any machine, protection based on trusted kernels managing capabilities becomes impossible. A malicious user could modify his kernel to subvert the capability checking and thereby bypass the protection scheme.

In [Donnelley 1981], a description is given of work being done at Lawrence Livermore Laboratory is given. Two schemes are described, one using a password in each capability, and one using public key cryptography. Although these schemes are similar to ours in some ways, they do not provide a way to protect individual rights bits to allow one capability to read an object and another to write it. Furthermore, our proposal addresses the problem of how to prevent users from impersonating servers or reading network traffic not intended for them. Both the F-boxes and the matrix method described in 2.4 can be used to fight wiretapping.

## 5. REFERENCES

Almes, G.T, Black, A.P., Lazowska, E.D., and Noe, J.D.: "The Eden System,: A Technical Review," *IEEE Trans Softw. Eng.* , vol. SE-11, pp. 43-59, Jan. 1985.

Birrell, A.D. and Nelson, B.J. "Implementing Remote Procedure Calls," *ACM Trans. Comp. Syst.*, vol. 2, pp. 39-59, Feb. 1984.

Dennis, J.B. and Van Horn, E.C. "Programming Semantics for Multiprogrammed Computations," *Comm. ACM*, vol. 9, pp. 143-154, March 1966.

Diffie, W., and Hellman, M.E.: "New Directions in Cryptography," *IEEE Trans. Inf. Theory*, vol. IT-22, pp. 644-654, Nov. 1976.

Donnelley, J.E., "Managing Domains in a Network Operating System," *Proc. Conf on Local Networks and Distributed Office Systems* , Online, pp. 345-361, 1981.

Evans, A., Kantrowitz, W., and Weiss, E: "A User Authentication Scheme not Requiring Security in the Computer," *Commun. ACM*, vol. 17, pp. 437-442, Aug. 1974.

Levy, H.: *Capability-Based Computer Systems,* Bedford, Mass.: Digital Press, 1984.

Mullender, S.J. "Principles of Distributed Operating System Design" Ph.D. thesis, Vrije Universiteit, Amsterdam, 1985.

Mullender, S.J. and Tanenbaum, A.S. "Protection and Resource Control in Distributed Operating Systems," *Computer Networks* (to appear in 1985).

Mullender, S.J., and Tanenbaum, A.S.: "A Distributed File Server Based on Optimistic Concurrency Control," Report IR-80, Wiskundig Seminarium, Vrije Universiteit, 32 pp. Nov. 1982.

Mullender, S.J., and Vitanyi, P.M.B. "Distributed Match-Making for Processes in

Computer Networks," Report CS-8424, Centrum v Wiskunde en Informatica, Dec. 1984.

Nelson, B.J. "Remote Procedure Call," Tech. Rep. CSL-81-9, Xerox PARC, 1981.

Purdy, G.B.: "A High-Security Log-in Procedure," *Commun. ACM*, vol. 17, pp. 442-445, Aug. 1974.

Rashid, R.F., and Robertson, G. G. "Accent: A Communication Oriented Network Operating System Kernel," *Proc. Eigth Symp. on Operating Syst. Prin.*, ACM, pp. 64-75, 1981.

Spector, A.Z. "Performing Remote Operations Efficiently on a Local Computer Network," *Comm. ACM.*, vol. 25, pp. 246-260, April 1982.

Tanenbaum, A.S., and Mullender, S.J. "The Design of a Capability-Based Distributed Operating System," *Computer Journal*, (to appear in 1985).

Wilkes, M.V.: *Time Sharing Computer Systems*, New York: American Elsevier, 1968.