Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

S.J. Mullender, A.S. Tanenbaum

The design of a capability-based distributed operating system

Department of Computer Science Report CS-R8418 October

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

The Design of a Capability-Based Distributed Operating System

Sape J. Mullender

Centre for Mathematics and Computer Science
Amsterdam, The Netherlands

Andrew S. Tanenbaum

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

Fifth generation computer systems will use large numbers of processors to achieve high performance. In this paper a capability-based operating system designed for this environment is discussed. Capability-based operating systems have traditionally required large, complex kernels to manage the use of capabilities. In our proposal, capability management is done entirely by user programs without giving up any of the protection aspects normally associated with capabilities. The basic idea is to use one-way functions and encryption to protect sensitive information. Various aspects of the proposed system are discussed.

1980 Mathematics Subject Classification: 68A05, 68B20.
1982 CR Categories: C.2.2, C.2.4, D.4.4, D.4.6.
Keywords & Phrases: distributed operating systems, capabilities, connectionless protocols, transaction-oriented protocols, protection, accounting, file systems, service model.

69 C22, 69 C24, 69 D54, 69 D56

1. Introduction

Fifth generation computers must be fast, reliable, and flexible. One way to achieve these goals is to build them out of a small number of basic modules that can be assembled together to realize machines of various sizes. The use of multiple modules can make the machines not only fast, but also achieve a substantial amount of fault tolerance. The system architecture and software for such machines are described below.

1.1. System Architecture

The price of processors and memory is decreasing at an incredible rate. Extrapolating from the current trend, it is likely that a single board containing a powerful CPU, a substantial fraction of a megabyte of memory, and a fast network interface will be available for a manufacturing cost of less than $100 in 1990. Our intention is therefore to do research on the architecture and software of machines built up of a large number of such modules.

In particular, we envision three classes of machines: (1) personal computers consisting of a high-quality bit-map display and a few processor-memory modules; (2) departmental machines consisting of hundreds of such modules; and (3) large mainframes consisting of thousands of them. The primary difference between these

Report CS-R8418
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

machines is the number of modules, rather than the type of the modules. In principle, any of these machines can be gracefully increased in size to improve performance by adding new modules or decreased in size to allow removal and repair of defective modules. The software running on the various machines should be in essence identical. Furthermore, it should be possible to connect different machines together to form even larger machines and to partition existing machines into disjoint pieces when necessary, all in a way transparent to the user level software.

This model is superior to the oft-proposed "Personal Computer Model," in a number of ways. In the personal computer model, each user has a dedicated minicomputer, complete with disks, in his office or at home. Unfortunately, when people work together on large projects, having numerous local file systems can lead to multiple, inconsistent copies of many programs. Also, the noise generated by disks in every office, and the maintenance problems generated by having machines spread all over many buildings can be annoying.

Furthermore, computer usage is very bursty: most of the time the user does not need any computing power, but once in a while he may need a very large amount of computing power for a short time (e.g., when recompiling a program consisting of 100 files after changing a basic shared declaration). The fifth generation computer we propose is especially well suited to bursty computation. When a user has a heavy computation to do, an appropriate number of processor-memory modules are temporarily assigned to him. When the computation is completed, they are returned to the idle pool for use by other users.

1.2. System Software

A machine of the type described above requires radically different system software than existing machines. Not only must the operating system effectively use and manage a very large number of processors, but the communication and protection aspects are very different from those of existing systems.

Traditional networks and distributed systems are based on the concept of two processes or processors communicating via connections. The connections are typically managed by a hierarchy of complex protocols, usually leading to complex software and extreme inefficiency. (An effective transfer rate of 0.1 megabit/sec over a 10 megabit/sec local network, which is only 1% utilization, is frequently barely achievable.)

We reject this traditional approach of viewing a distributed system as a collection of discrete processes communicating via multilayer (e.g., ISO) protocols, not only because it is inefficient, but because it puts too much emphasis on specific processes, and by inference, on processors. Instead we propose to base the software design on a different conceptual model – the object model. In this model, the system deals with abstract objects, each of which has some set of abstract operations that can be performed on it.

Associated with each object are one or more "capabilities" [Dennis66] which are used to control access to the object, both in terms of who may use the object and what operations he may perform on it. At the user level, the basic system primitive is performing an operation on an object, rather than such things as establishing connections, sending and receiving messages, and closing connections. For example, a typical object is the file, with operations to read and write portions of it.

The object model is well-known in the programming languages community under the name of "abstract data type" [Liskov74]. This model is especially well-suited to a distributed system because in many cases an abstract data type can be implemented on one of the processor-memory modules described above. When a user process executes one of the visible functions in an abstract data type, the system arranges for the necessary underlying message transport from the user's machine to that of the abstract data type and back. The header of the message can specify which operation is to be performed on which object. This arrangement gives a very clear separation between users and objects, and makes it impossible for a user to directly inspect the representation of an abstract data type by bypassing the functional interface.

A major advantage of the object or abstract data type model is that the semantics are inherently location independent. The concept of performing an operation on an object does not require the user to be aware of where objects are located or how the communication is actually implemented. This property gives the system the possibility of moving objects around to position them close to where they are frequently used. Furthermore, the issue of how many processes are involved in carrying out an operation, and where they are located is also hidden from the user.

It is frequently convenient to *implement* the object model in terms of clients (users) who send messages to services. A service is defined by a set of commands and responses. Each service is handled by one or more server processes that accept messages from clients, carry out the required work, and send back replies. The design of these servers and the design of the protocols they use form an important part of the system software of our proposed fifth generation computers.

As an example of the problems that must be solved, consider a file server. Among other design issues that must be dealt with are how and where information is stored, how and when it is moved, how it is backed up, how concurrent reads and writes are controlled, how local caches are maintained, how information is named, and how accounting and protection are accomplished. Furthermore, the internal structure of the service must be designed: how many server processes are there, where are they located, how and when do they communicate, what happens when one of them fails, how is a server process organized internally for both reliability and high performance, and so on. Analogous questions arise for all the other servers that comprise the basic system software.

2. Communication Primitives and Protocols

In the literature about computer networks, one finds much discussion of the ISO OSI reference model [Tanenbaum81] these days. It is our belief that the price that must be paid in terms of complexity and performance in order to achieve an "open" system in the ISO sense is much too high, so we have developed a much simpler set of communication primitives, which we will now describe.

2.1. Transaction vs. Stream Communication

Most distributed systems have a connection mechanism that is based on the idea of two processes going to some effort to set up a connection, using the connection, and then tearing it down. The assumption is that a connection will be used for a stream of information so long that the overhead needed to set it up and tear it down are basically negligible. Most streams will consist of a file of one kind or another – a source program, a binary program, an input file, and so on. To see how long the average file is, we have conducted some measurements on the UNIX† system used in our department by the faculty and staff for research (no students, thus). The results of these measurements show that 34% of all files are less than 512 bytes, 52% are less than 1K bytes, 67% are less than 2K bytes, 79% are less than 4K bytes, 88% are less than 8K bytes, and 94% are less than 16K bytes.

The above considerations have led us to a different approach [Mullender84]. With packets of even 2K bytes, two thirds of all files fit into a single packet. Consequently, it is much simpler to adopt a "Request-Reply" or "Transaction" style of communication, in which the basic primitive is the client sending a request to a server and the

† UNIX is a Trademark of Bell Laboratories.

server sending a reply back to the client. The client uses `trans` and the server `getreq` and `putrep`. `Trans` sends a request, and blocks until a reply is received. `Getreq` blocks the server until a request is received, which can then be processed, after which a reply can be sent using `putrep`. Each request-reply pair is completely self-contained, and independent of any other ones that may previously been sent. In other words, no concept of a "connection" exists. Not only is this conceptually much more appropriate for use in an operating system, but it is much simpler to implement than a complex 7-layer protocol, not to mention offering lower delay. Henceforth we will refer to a request-reply pair as a *transaction*, which is not to be confused with transactions with a data base.

2.2. Basic Communication Protocol

Instead of a 7-layer protocol, we effectively have a 4-layer protocol. The bottom layer is the Physical Layer, and deals with the electrical, mechanical and similar aspects of the network hardware. The next layer is the Port Layer, and deals with the location of services, the transport of (32K byte) datagrams (packets whose delivery is not guaranteed) from source to destination and enforces the protection mechanism, which will be discussed in the next section. On top of this we have a layer that deals with the reliable transport of bounded length (32K byte) requests and replies between client and server. We have called this layer the Transaction Layer. The final layer has to do with the semantics of the requests and replies, for example, given that one can talk to the file server, what commands does it understand. The bottom three layers (Physical, Port and Transaction) are implemented by the kernel and hardware; only the Transaction Layer interface is visible to users.

Since systems of the kind we are describing will use high-speed, highly reliable local networks, few, if any, of the complex mechanisms designed for flow- and error-control in long-haul networks are useful here. Among other things, a simple stop-and-wait protocol is sufficient. The main function of the Transaction Layer is to provide an end-to-end *message* service built on top of the underlying *datagram* service, the main difference being that the former uses timers and acknowledgements to guarantee delivery whereas the latter does not.

The Transaction Layer protocol is straightforward. When the client does a `trans`, a packet, or sequence of packets, containing the request is sent to the server, the client is blocked, and a timer is started (inside the Transaction Layer). If the server does not acknowledge receipt of the request packet before the timer expires (usually by sending the reply, but in some special cases by sending a separate acknowledgement packet), the Transaction Layer retransmits the packet again and restarts the timer. When the reply finally comes in, the client sends back an acknowledgement (possibly piggybacked onto the next request packet) to allow the server to release any resources, such as buffers, that were acquired for this transaction. Under normal circumstances, reading a long file, for example, consists of the sequence

From client: request for block 0
From server: here is block 0
From client: acknowledgement for block 0 and request for block 1
From server: here is block 1
etc.

The protocol can handle the situation of a server crashing and being rebooted quite easily since each request contains the capability for the file to be read and the position in the file to start reading. Between requests, the server has no "activation record" or other table entry whose loss during a crash causes the server to forget which files were open, etc., because no concept of an open file or a current position in a file exists on the server's side. Each new request is completely self-contained. Of course for efficiency reasons, a server may keep a cache of frequently accessed i-nodes, file blocks etc., but these are not essential and their loss during a crash will merely slow the server down slightly while they are being dynamically refreshed after a reboot.

2.3. The Port Layer

The Port Layer is responsible for the speedy transmission of 32K byte datagrams. The Port Layer need only do this reasonably reliably, and does not have to make an effort to guarantee the correct delivery of every datagram. This is the responsibility of the Transaction Layer. Our results show that this approach leads to significantly higher transmission speeds, due to simpler protocols.

Theoretically, very high speeds are achievable in modern local-area networks. A typical speed for DMA transfers is 1 byte/μsec, and the typical transmission speed of a 10 Mbit local-area network is also 1 byte/μsec. Since DMA transfer and network transfer cannot overlap, but DMA at the destination host *can* overlap with the DMA of the next packet at the source host, an upper bound for the transfer rate of a typical local-area network is 500,000 bytes/sec point-to-point.

In practise, however, speeds of 100,000 bytes per second between user processes have rarely been achieved. Obviously, to achieve higher transmission rates, the overhead of the protocol must be kept very low indeed, while an effort must be made to overlap DMAs at both communicating parties. To achieve this, we have chosen a large datagram size for the Port Layer, which has to split up the datagrams into small packets that the network hardware can cope with. This approach allows the implementor of the Port Layer to exploit the possibilities that the hardware has to offer to achieve an efficient stream of packets.

Our implementation of the Port Layer interfaces to a 10 Mbit token ring that allows *scatter-gather*; that is, a packet can be sent to or from the interface in several DMA transfers, and then transmitted over the network separately. This allows us to do two important things to speed up the protocol. First, when a packet is received, the header can be inspected separately, so the protocol can decide where in memory the packet must go. The protocol driver can then transfer the packet directly from the interface to the right place in memory, without having to copy it. A copy loop would halve the transmission speed. Second, the separation of DMA and transmission allows the driver to prepare a transmission by doing the DMA. The transmission can then be initiated immediately when the signal is received that the receiver is ready. In our implementation of the Port Layer, these considerations have resulted in the protocol that will now be described.

The transmitter begins by transferring and sending the first 2K of the datagram to be transmitted (2K is the maximum packet size allowed by the hardware). Immediately after the transmission is complete, the DMA for the next 2K bytes is started, but they are not yet transmitted. In the mean time, the receiver is interrupted by the arrival of the first packet. It extracts the header, examines it and decides where the body of the packet should go. Then the body of the packet is transferred from the interface to its final location in memory. While this is being done, the receiver prepares a tiny *acknowledgement* packet to tell the transmitter it is prepared for the next packet. As soon as the DMA transfer of the previous packet has finished, this acknowledgement is sent back to the transmitter. When the transmitter receives it, the transfer of the next packet to the interface will have finished, so it can then be sent immediately. This sequence is continued until the whole datagram is transmitted.

2.4. The Transaction Layer

It is the responsibility of the Transaction Layer to guarantee the arrival of requests and replies. The Transaction Layer makes use of the Port Layer and timers to achieve this.

The interface to the transaction layer basically consists of three calls, one for clients, and two for servers. All calls use a small datastructure, called `Mref`, which contains a pointer to a small fixed-size out-of-band buffer for the transmission of commands and parameters to the server, a pointer to the main body of data to be transferred, and the length of the main body of data (0 to 32768), as follows:

```
typedef struct Mref {
        char     *M_oob;
        char     *M_buf;
        unsigned  M_len;
} Mref;

typedef struct Cap {
        Port      C_port;           /* 6-byte port */
        char      C_private[10];    /* 10-byte private */
} Cap;    /* capability */
```

The client, in order to do a transaction calls

```
trans(cap, req, rep);
        Cap *cap;
        Mref *req, *rep;
```

The server receives requests and sends replies with

```
getreq(port, cap, req);
        Port *port;
        Cap *cap;
        Mref *req;

putrep(rep);
        Mref *rep;
```

In principle, the Transaction Layer works as follows: When a client calls `trans`, the Transaction Layer generates a *reply-port* to enable the server to send a reply. The server port is deduced from the capability; the first 48 bits of the capability for an object identify the service that controls the object. The request is then sent, using `put`, and a *retransmission timer* is started.

The server, which previously had made a call to `getreq`, receives the request; the capability is filled in, and the received message is put in the buffers referred to by `req`. As soon as the request is received, the server's Transaction Layer starts a *piggy-back timer*. When the server has not sent a reply before this timer expires, a separate acknowledgement is sent to put the client at ease, and stop its retransmission timer. When the server sends a reply to the client the same thing happens, more or less, with the role of client and server reversed. When a client makes a sequence of transactions with a single server, a subsequent request will acknowledge receipt of the previous reply.

The client maintains one more timer, the *crash timer*. This timer is set when the server's acknowledgement to a request has been received, and is used to detect server crashes. Whenever this timer expires, the client sends an "are you still alive?" packet to the server, to which the server replies with an acknowledgement.

When transactions occur quickly, one after the other, no extra acknowledgements are sent at all. Only when transactions take a long time (say, longer than a minute), acknowledgements are sent, and when transactions take much longer than that (say, ten minutes) then "are you still alive" messages begin to be sent.

2.5. *Timer Management*

If the timers are started and stopped in exactly the way described above, the Transaction Layer would become unacceptably slow. Per (quick) transaction, two retransmission timers and two piggyback timers would have to be started and stopped, eight timer actions altogether.

There is a much more efficient way of dealing with timers, one that makes use of a *sweep algorithm*. This algorithm does not implement very accurate timers, but accuracy of the timer intervals is not very important to the correct and efficient operation of the protocol.

The sweep algorithm is run every N clock ticks. N must be chosen such that N ticks is about the minimum timer interval needed (the piggyback timer interval). Whenever the algorithm is called, it makes a sweep over all outstanding transactions. If the state of a transaction has changed, the new state is recorded. If it has not changed, a counter is incremented, telling for how long the state has remained the same. If the (state, counter) combination has reached a certain value, the sweep algorithm carries out the appropriate actions, usually sending an acknowledgement, retransmitting a message, or aborting a transaction.

Because this algorithm is used there is no code needed in the transaction code itself, reducing the overhead of the Transaction Layer significantly. In this way, the code executed in the Transaction Layer is optimised for the normal case (no errors).

2.6. *Blocking vs. Non-Blocking Transaction Primitives*

Most services need to be able to handle multiple requests from different clients simultaneously. It therefore seems natural to implement non-blocking calls for interprocess communication, as this will allow a service to react to events in the order they occur. When blocking communication calls are used, a server is forced to wait for the specific event that unblocks the call.

Because it is rather difficult to write correct code for a process which has to handle multiple flows of control indeterministically, the ***Amoeba*** system provides the concept of *tasks*, sharing an address space. A number of tasks in one address space forms a *cluster*, and specific rules govern the scheduling of tasks within a cluster: only one task can run at a time, and a task runs until it voluntarily relinquishes control (e.g., on `trans` and `getreq` calls).

A server can thus easily be structured as a collection of co-operating tasks, each task handling one request. This model has greatly simplified the structure of services, as each task making up the server cluster now has a single thread of execution. The model also obviated the need for non-blocking transaction calls, with their complicated (and slow) extra interface for handling interrupts.

2.7. *Results*

Two versions of the algorithm have now been implemented. The one described has been implemented on the ***Amoeba*** distributed operating system, and achieves over 300,000 bytes a second from user process to user process (using M68000s and a Pronet* ring). It is now being implemented under UNIX where we expect to obtain more than 200,000 bytes/sec, assuming the communicating processes are not swapped.

* PRONET is a trademark of Proteon Associates, Inc.

An older version of the protocol, using 2K byte datagrams, now gets 90,000 bytes/sec across the network between two VAX-750s running a normal load of work, without causing a significant load on the system itself.

Several services, implemented under UNIX, are using the Transaction Layer interface, and it is our experience that these services are easy to design and that they work efficiently.

3. Ports and Capabilities

3.1. Ports

Every service has one or more *ports* [Mullender82b] to which client processes can send messages to contact the service. Ports consist of large numbers, typically 48 bits, which are known only to the server processes that comprise the service, and to the service's clients. For a public service, such as the system file service, the port will be generally 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 public file service may refuse to read or write files for clients lacking account numbers, appropriate authorization, etc.

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. The basic primitive operations offered by the system are `trans`, `putreq` and `getrep`. Since everyone knows the port of the file server, as an example, how does one insure that malicious users do not execute `getreq`s on the file server's port, in effect impersonating 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 `getreq` from which port. We reject this strategy because some machines, e.g., personal computers connected to larger multimodule systems may not be trustworthy, 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. First we will describe the hardware solution; later we will describe the software solution.

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 a trusted operating system running in kernel mode, it can also be put into operating system software. In any event, we assume that somehow or other all packets 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 [Wilkes68, Purdy74, Evans74] 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 so difficult that the only approach is to try every possible G to see which one produces P. If P and G contain sufficient bits, this approach can be made to take millions of years on the world's largest supercomputer, thus making it effectively impossible to find G given only P. Note that a one-way function differs from a cryptographic transformation in the sense that the latter must have an inverse to be useful, but the former has been carefully chosen so that no inverse can be found.

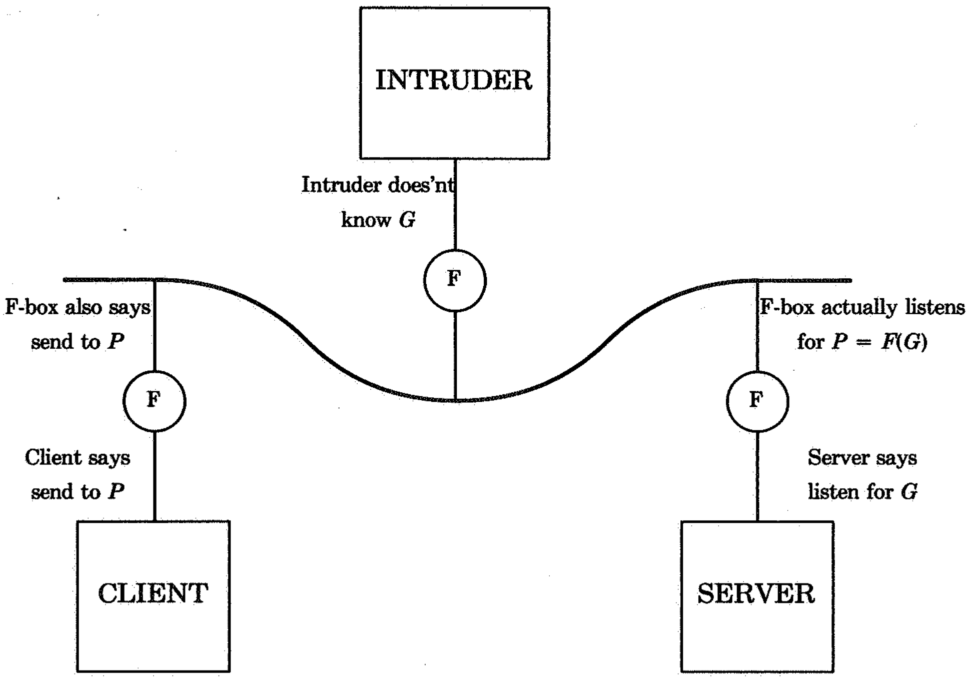
INTRUDER
Intruder does'nt know G
F
F-box also says send to P
F
Client says send to P
CLIENT
F-box actually listens for $P = F(G)$
F
Server says listen for G
SERVER

FIGURE 1.

Using the one-way F-box, the server authentication can be handled in a simple way, illustrated in FIGURE 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 `getreq(`G`, cap, req)`. The F-box then computes $P = F(G)$ and waits for packets containing P to arrive. When one arrives, it is given to the appropriate process. To send a packet to the server, the client merely does `trans(cap, req, rep)`, where the *port* field of `cap` is set to P. This will cause a datagram to be sent by the local F-box with P in the destination-port field of the header. The F-box on the sender's side does not perform any transformation on the P field of the outgoing packet.

Now let us consider the system from an intruder's point of view. To impersonate a server, the intruder must do `getreq(`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 packets not intended for him. Replies from the server to the client are protected the same way, only with the client's Transaction Layer picking a get-port for the reply, say, G', and including $P' = F(G')$ in the request packet.

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 packet presented to the F-box 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 packets for which the corresponding `getreq` 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 with precluding many as-yet-unthought-of operating systems to be designed in the future.

3.2. 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 service, which is a user (as opposed to kernel) program, and which understands the capabilities for its objects.

SERVER	OBJECT	RIGHTS	RANDOM

FIGURE 2.

A capability typically consists of four fields, as illustrated in FIGURE 2:

1. The put-port of the service that manages the object
2. An Object Number meaningful only to the service managing the object
3. A Rights Field, which contains a 1 bit for each permitted operation
4. A Random Number for protecting each object

The basic model of how capabilities are used can be illustrated by a simple example: a client wishes to create a file using the file service, 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 service's put-port specifying that a file is to be created. The request might contain a file name, account number and similar attributes, depending on the exact nature of the file service. 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 message containing the capability and some data. When the `write` request arrived at the file server process, the server would normally use the object number contained in the capability as as index into its tables to locate the object. For a UNIX like file server, the object number would be the i-node 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 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, it can easily be modified to provide that distinction. In the modified 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 (e.g., 8 bits), which is initially all 1s indicating that all operations are legal, and the Random Number Field (e.g., 56 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 number (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 Number 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.

When this modified protection system is used, the owner of the object can easily give an exact copy of the capability to another process by just sending it the bit pattern, but to pass, say, read-only access, is 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 whose Rights Field is the Boolean-and of the Rights Field in the capability and the bit mask. By choosing the bit mask carefully, the capability owner can mask out any operations that the recipient is not permitted to carry out.

This modified system works well except that it requires going back to the server every time a sub-capability with fewer rights is needed. We have devised yet another protection system that does not have this drawback. This third scheme requires the use of 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 the file table and the Random Number Field, just as in the first scheme presented. It also sets all the Rights Field bits to 1.

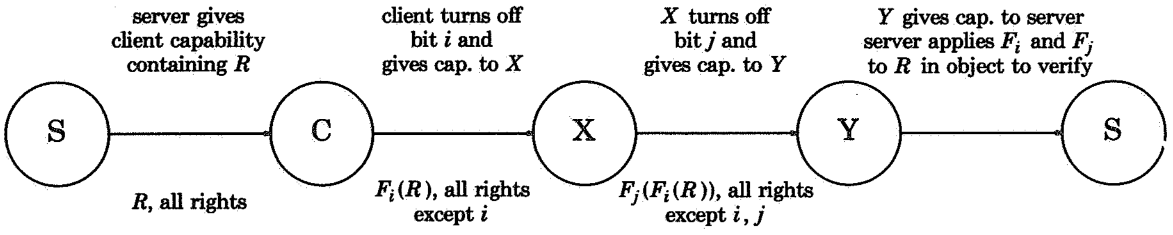
server gives client capability containing R
client turns off bit i and gives cap. to X
X turns off bit j and gives cap. to Y
Y gives cap. to server server applies F_i and F_j to R in object to verify
S
C
X
Y
S
R, all rights
$F_i(R)$, all rights except i
$F_j(F_i(R))$, all rights except i, j

FIGURE 3.

A client can delete permission k from a capability by replacing the random number, 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 the file 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. The mechanism is illustrated in FIGURE 3. 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 immediately. 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.

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 retract existing capabilities. All that the owner of an object need do is ask the server to change the random number stored in the file table. Obviously this operation must be protected with a bit in the Rights Field, but if it succeeds, all existing capabilities are instantly invalidated.

3.3. Protection without F-Boxes

Earlier we said that protection could also be achieved without F-boxes. It is slightly more complicated, since it uses both conventional and public-key encryption, 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 packet 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 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 packet. The data need not be encrypted, although that is also possible if needed.

Each machine is assumed to know its row and column of the matrix, and nothing else (how this will be achieved will be discussed shortly). With this arrangement, intruder I can easily capture packets 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, that is, to contain random numbers that agree with those stored in the file tables.

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 packet containing both K and a newly chosen conventional key to be used for reverse traffic. This packet 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 packet contains K, C can be sure that the other conventional key was indeed generated by the 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 packets.

4. The Amoeba File System

The file system has been designed to be highly modular, both to enhance reliability and to provide a convenient testbed for doing research on distributed file systems. It consists of three completely independent pieces: the block service, the file service, and the directory service. In short, the block service provides commands to read and write raw disk blocks. As far as it is concerned, no two blocks are related in any way, that is, it has no concept of a file or other aggregation of blocks. The file service uses the block service to build up files with various properties. Finally, the directory service provides a mapping of symbolic names onto object capabilities.

4.1. Block Service

The block service is responsible for managing raw disk storage. It provides an object-oriented interface to the outside world to relieve file servers from having to understand the details of how disks work. The principle operations it performs are:

- `allocate` a block, write data into it, and return a capability to the block
- given a capability for a block, `free` the block
- given a capability for a block, `read` and return the data contained in it
- given a capability for a block and some data, `write` the data into the block
- given a capability for a block and a key, `lock` or unlock the block

These primitives provide a convenient object-oriented interface for file servers to use. In fact, any client who is unsatisfied [Stonebraker81, Tanenbaum82] with the standard file system can use these operations to construct his own.

The first four operations of `allocate`, `free`, `read`, and `write` hardly need much comment. The fifth one provides a way for clients to lock individual blocks. Although this mechanism is crude, it forms a sufficient basis for clients (e.g., file systems) to construct more elaborate locking schemes, should they so desire. One other operation is worth noting. The data within a block is entirely under the control of the processes possessing capabilities for it, but we expect that most file servers will use a small portion of the data for redundancy purposes. For example, a file server might use the first 32 bits of data to contain a file number, and the next 32 bits to contain a relative block number within the file. The block server supports an operation `recovery`, in which the client provides the account number it uses in `allocate` operations and requests a list of all capabilities on the whole disk containing this account number. (The block server stores the account number for each block in a place not accessible to clients.) Although `recovery` is a very expensive operation, in effect requiring a search of the entire disk, armed with all the capabilities returned, a file server that lost all of its internal tables in a crash could use the first 64 bits of each block to rebuild its entire file list from scratch.

4.2. File Service

The purpose of splitting the block service and file service is to make it easy to provide a multiplicity of different file services for different applications. One such file service that we envision is one that supports flat files with no locking, in other words, the UNIX model of a file as a linear sequence of bytes with no internal structure and essentially no concurrency control. This model is quite straightforward and will therefore not be discussed here further.

A more elaborate file service with explicit version and concurrency control for a multiuser environment will be described instead [Mullender82a]. This file service is designed to support data base services, but it itself is just an ordinary, albeit slightly advanced, file service. The basic model behind this file service is that a file is a time-ordered sequence of versions, each version being a snapshot of the file made at a moment determined by a client. At any instant, exactly one version of the file is the *current version*. To use a file, a client sends a message to a file server process containing a file capability and a request to create a new, private version of the current version. The server returns a capability for this new version, which acts like it is a block for block copy of the current version made at the instant of creation. In other words, no matter what other changes may happen to the file while the client is using his private version, none of them are visible to him. Only changes he makes himself are visible.

Of course, for implementation efficiency, the file is not really copied block for block. What actually happens is that when a version is created, a table of pointers (capabilities) to all the file's blocks is created. The capability granted to the client for the new version actually refers to this version table rather than the file itself. Whenever the client reads a block from the file, a bit is set in the version table to indicate that the corresponding block has been read. When a block is modified in the version, a new block is allocated using the block server, the new block replaces the original one, and its capability is inserted into the version table. A bit indicating that the block is a new one rather than an original is also set. This mechanism is sometimes called "copy on write."

Versions that have been created and modified by a client are called *uncommitted versions*. At a particular moment, the current version may have several (different) uncommitted versions derived from it in use by different clients. When a client is finished modifying his private version, he can ask the file server to *commit* his version, that is, make it the current version instead of the then current version. If the version from which the to-be-committed version was derived is still current at the time of the commit, the commit succeeds and becomes the new current version.

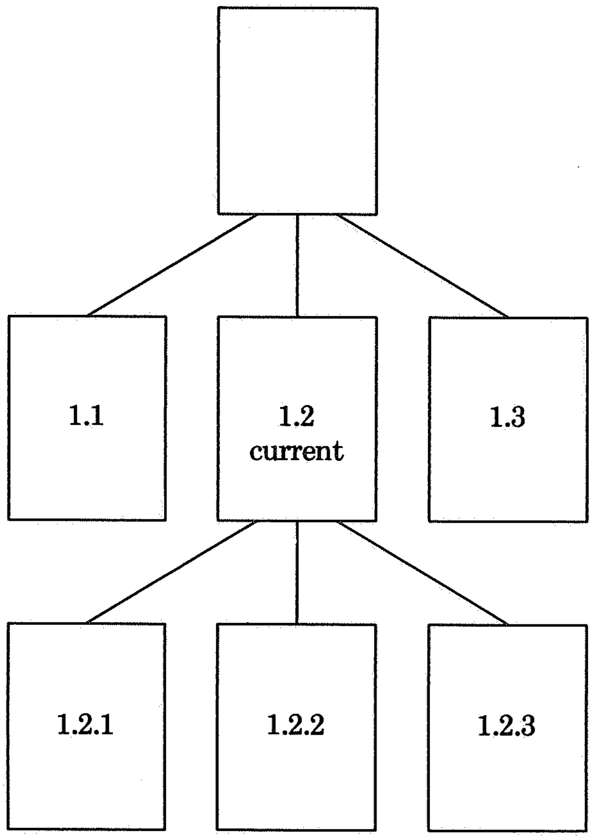

FIGURE 4.

As an example, suppose version 1 is initially the current version, with various clients creating private versions 1.1, 1.2, and 1.3 based on it. If version 1.2 is the first to commit, it wins and 1.2 becomes the new current version, as illustrated in FIGURE 4. Subsequent requests by other clients to create a version will result in versions 1.2.1, 1.2.2, and 1.2.3, all initially copies of 1.2.

The fun begins when the owner of version 1.3 now tries to commit. Version 1, on which it is based, is no longer the current version, so a problem arises. To see how this should be handled, we must introduce a concept from the data base world, *serializability* [Eswaran76, Papadimitriou79]. Two updates to a file are said to be serializable if the net result is either the same as if they were run sequentially in either order. As a simple example, consider a two character file initially containing "ab." Client 1 wants to write a "c" into the first character, wait a while, and then write a "d" into the second character. Client 2 wants to write an "e" into the first character, wait a while, and then write an "f" into the second character. If 1 runs first we get "cd"; if 2 runs first we get "ef." Both of these are legal results, since the file server cannot dictate when the users run. However, its job is to prevent final configurations of "cf" or "de," both of which result from interleaving the requests. If a client locks the file before starting, does all its work, and then unlocks the file, the result will always be either "cd" or "ef," but never "cf" or "de." What we are trying to do is accomplish the same goal without using locking.

The idea behind not locking is that most updates, even on the same file, do not affect the same parts of the file, and hence do not conflict. For example, changes to an airline reservation data base for flights from San Francisco to Los Angeles do not conflict with changes for flights from Amsterdam to London. The strategy behind our commit mechanism is to let everyone make and modify versions at will, with a check for serializability when a commit is attempted. This mechanism has been proposed for data base systems [Kung81], but as far as we know, not for file systems.

The serializability check is straightforward. If a version to be committed, A, is based on the version that is still current, B, it is serializable and the commit succeeds. If it is not, a check must be made to see if all of the blocks belonging to A that the client has read are the same in the current version as they were in the version from which A was derived. If so, the previous commit or commits only changed blocks that the client trying to commit A was not using, so there is no problem and the commit can succeed.

If, however, some blocks have been changed, modifications that A's owner has made may be based on data that are now obsolete, so the commit must be refused, but a list is returned to A's owner of blocks that caused conflicts, that is, blocks marked "read" in A and marked "written" in the current version (or any of its ancestors up to the version on which A is based). At this point, A's owner can make a new version and start all over again. Our assumption is that this event is very unlikely, and that is occasional occurrence is a price worth paying for not having locking, deadlocks, and the delays associated with waiting for locks.

4.3. Directory Service

Because it is frequently inconvenient to deal with long binary bit strings such as capabilities, a directory service is needed to provide symbolic naming. The directory service's task is to manage directories, each of which contains a collection of (ASCII name, capability) pairs. The principal operation on a directory object is for a client to present a capability for a directory and an ASCII name, and request the directory service to look up and return the capability associated with the ASCII name. The inverse operation is to store an (ASCII name, capability) pair in a directory whose capability is presented.

5. Process Management

Like any other operating system, this one must also have a way to manage processes. In our design, processes are created and managed by the process service, which consists of three major subsystems, the generic server, the process server, and the boot server.

5.1. Generic Server

The idea behind the generic server is that much of the time a user wants a certain program to be run, but does not care about where it is run or on which CPU type. For example, a user might have a Pascal program to be compiled, and wants a Pascal compiler that produces, say, Motorola 68000 code. However, he does not care whether the compiler itself runs on a 68000, a VAX or any other CPU. We speak of this as a generic Pascal compiler.

The generic server's job is to locate a suitable hardware/software combination and start it up. This can be done by maintaining internal tables of locations where the appropriate service is likely to be located. By sending a message to the chosen service, the generic server can see if the corresponding server is currently available and willing to take on the offered work. If so, it can begin; if not, the generic server can broadcast a request for bids to see if someone else can be located. If no willing server exists, the generic server will have to cause one to be created by invoking the process server.

5.2. Process Server

The process server's job is to take a process descriptor sent to it, locate a free processor, and send sufficient information to the processor to allow the processor to run. The process descriptor must contain at least the following information:

1. The CPU type desired.
2. A capability for the binary file to be executed.
3. Capabilities for process environment.
4. Accounting information.

The CPU type and binary file capability are obvious. The third item has to do with things like the file descriptors and environment strings in UNIX. When a UNIX process is started up, it inherits certain parameters from its parent, among these are usually file descriptors for standard input, output, and diagnostic, and possibly other files as well. In our design, a process can inherit capabilities for standard input, standard output, and standard diagnostic, as well as other ones. By using these, one can implement UNIX pipes and filters easily, as well as more general mechanisms (e.g., passing capabilities to third parties, storing them in files for later use, etc.).

Another area that the process service must deal with is scheduling. It must allocate processes to processors, and possibly control migration and swapping among processors as well. By introducing the concept of a "process image" which contains all the information necessary to run a process (e.g., its memory, registers, capabilities, etc.) it becomes straightforward to handle process migration and swapping in a unified way. When a process is swapped out to a disk somewhere, there is no need to have it swapped back to the same machine that it originated on.

5.3. Boot Service

Many services must achieve high availability. Our approach to this issue is using fault tolerance, rather than fault intolerance. In the former, one expects hardware and software to fail, and makes provision for dealing with it; in the latter, one assumes that they are perfect and that no such provision need be made. Since many services are faced with the same problem: how to provide high availability in the face of occasional crashes, we have abstracted out a common part of the crash recovery mechanism and put it into a separate service, the boot service.

Any service that wants to provide a continuous availability can register with the boot service. Such registration entails providing a polling message to send the service periodically, the expect reply, the polling frequency, and a prescription of what to do in case of failure. The boot service then sends the polling message to the service at the requested frequency. As long as the service continues to send the appropriate reply, all is well and the boot service has nothing else to do.

However, if the service fails to reply properly, or fails to reply at all within an agreed upon time interval, the boot service declares the service to be out-of-order, and goes to the process service to start up a new version of it. Of course, the boot service itself must not crash, but it consists of a number of server processes that constantly check each other, and if need be, replace sick members with healthy ones.

6. Resource Management

In keeping with our general philosophy of making the system kernel as small as possible, we have devised a way to put the resource control and accounting outside the kernel. Furthermore, a clear distinction is made between policy and mechanism, so that subsystem designers can implement their own policies with the standard mechanisms.

Traditionally, accounting was used by the management of a computer center to levy charges for the use of the computer center's resources: CPU time, file space, lineprinter paper. This method worked quite well in the past, when hardware resources were expensive compared to the software used. Nowadays, hardware is cheap, software expensive. However, in the traditional approach there is usually no possibility to bill users for the use of a particular piece of software, or to have one user bill another for using his services.

Additionally, distributed systems need not be under control of one centralized management any more; private, personal computers can be plugged into the network and both use and offer services to the rest of the network. The accounting mechanisms in a distributed systems must be able to handle this new view on operating systems and allow any user that sets up a service to gather information about who uses his service.

6.1. Bank Service

The bank service is the heart of the resource management mechanism. It implements an object called a "bank account" with operations to transfer virtual money between accounts and to inspect the status of accounts. Bank accounts come in two varieties: individual and business. Most users of the system will just have one individual account containing all their virtual money. This money is used to pay for CPU time, disk blocks, typesetter pages, and all other resources for which the service owning the resource decides to levy a charge.

Business accounts are used by services to keep track of who has paid them and how much. Each business account has a subaccount for each registered client. When a client transfers money from his individual account to the service's business account, the money transferred is kept in the subaccount for that client, so the service can later ascertain each client's balance. As an example of how this mechanism works, a file service could charge for each disk block written, deducting some amount from the client's balance. When the balance reached zero, no more blocks could be written. Large advance payments and simple caching strategies can reduce the number of messages sent to a small number.

Another aspect of the bank service is its maintenance of multiple currencies. It can keep track of say, virtual dollars, virtual yen, virtual guilders and other virtual currencies, with or without the possibility of conversion among them. This feature makes it easy for subsystem designers to create new currencies and control how they are allocated among the subsystems users.

6.2. *Accounting Policies*

The bank service described above allows different subsystems to have different accounting policies. For example, a file or block service could decide to use either a buy-sell or a rental model for accounting. In the former, whenever a block was allocated to a client, the client's account with the service would be debited by the cost of one block. When the block was freed, the account would be credited. This scheme provides a way to implement absolute limits (quotas) on resource use. In the latter model, the client is charged for rental of blocks at a rate of X units per kiloblock-second or block-month or something else. In this model, virtual money is constantly flowing from the clients to the servers, in which case clients need some form of income to keep them going. The policy about how income is generated and dispensed is determined by the owner of the currency in question, and is outside the scope of the bank server.

7. Summary

This paper has discussed a model for a fifth generation computer system architecture and its operating system. The operating system is based on the use of objects protected by sparse capabilities. An outline of some of the key services has been given, notably the block, file, directory, generic, process, boot and bank services.

References

[Dennis66]

Dennis, J. B. and Horn, E. C. van, "Programming Semantics for Multiprogrammed Computations," *Comm. ACM*, vol. 9, no. 3, pp.143-155, March 1966.

[Eswaran76]

Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., "The Notions of Consistency and Predicate Locks in a Database Operating System," *Comm. ACM*, vol. 19, no. 11, pp.624-633, November 1976.

[Evans74]

Evans, A., Kantrowitz, W., and Weiss, E., "A User Authentication Scheme Not Requiring Secrecy in the Computer," *Comm. ACM*, vol. 17, no. 8, pp.437-442, August 1974.

[Kung81]

Kung, H. T. and Robinson, J. T., "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp.213-226, June 1981.

[Liskov74]

Liskov, B. and Zilles, S., "Programming With Abstract Data Types," *SIGPLAN Notices*, vol. 9, pp.50-59, April 1974.

[Mullender82a]

Mullender, S.J. and Tanenbaum, A.S., "A Distributed File Server Based on Optimistic Concurrency Control", IR-80, Vrije Universiteit, Amsterdam, November 1982.

[Mullender82b]

Mullender, S.J. and Tanenbaum, A.S., "Protection and Resource Control in Distributed Operating Systems", IR-79 (to appear in Computer Networks), Vrije Universiteit, Amsterdam, August 1982.

[Mullender84]
Mullender, S. J., Renesse, R. van, and Tanenbaum, A. S., "A Transaction-Oriented Transport Protocol", Report CS-R84XX, Centre for Mathematics and Computer Science (CWI), Amsterdam, November 1984.

[Papadimitriou79]
Papadimitriou, C. H., "Serializability of Concurrent Updates," *J. ACM*, vol. 26, no. 4, pp.631-653, October 1979.

[Purdy74]
Purdy, G. B., "A High Security Log-in Procedure," *Comm. ACM*, vol. 17, no. 8, pp.442-445, August 1974.

[Stonebraker81]
Stonebraker, M., "Operating System Support for Database Management," *Comm. ACM*, vol. 24, no. 7, pp.412-418, July 1981.

[Tanenbaum81]
Tanenbaum, A.S., "The ISO-OSI reference model", IR-71, Vrije Universiteit, Amsterdam, 1981.

[Tanenbaum82]
Tanenbaum, A. S. and Mullender, S. J., "Operating System Requirements for Distributed Data Base Systems," pp. 105-114 in Distributed Data Bases, ed. H. J. Schneider, North-Holland Publishing Co. (1982).

[Wilkes68]
Wilkes, M. V., *Time-Sharing Computer Systems.* New York:American Elsevier, 1968.