The Amoeba Distributed Operating System
(Part 1)

Sape J. Mullender
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The Amoeba Project is a distributed project on distributed operating systems. The project, which started as the author's PhD research project in 1978 [8], is now a joint project of CWI and Vrije Universiteit in Amsterdam. About a dozen people are working on the project, led by prof. dr. A. S. Tanenbaum (VU) and the author (CWI). This article describes the interprocess communication facilities and protection mechanisms of the Amoeba system. Part 2, which will appear in the next Newsletter, will be devoted to the services provided by the Amoeba Distributed Operating System.

1. Introduction

Distributed information processing has long been practised by living organisms. The human brain, one of the most complicated living organs, functions in a highly distributed manner; different parts of the brain have specialised to perform different functions, such as speech and vision. Yet there does not seem to be any central control in the brain, 'consciousness' cannot be pinpointed to one specific group of brain cells. Not a single function of the brain seems to be impaired when any cell in the brain dies. The individual cells in living organisms die, are replaced by others, and yet, the organism as a whole continues to function uninterruptedly. Most life forms use distributed control of some form or another. Even simple life forms, such as the one-celled *amoebæ*—which have no single 'command centre' to decide where to go and how to get there—are somehow capable of co-ordinated action.

Imitating nature in all aspects, man has finally begun to incorporate the principles of distributed information processing in his most complicated artifacts, computers. In their desire to construct better, faster and more reliable information processing systems, researchers are building networks of many computers which co-operate to preform their task more quickly and more reliably.

The technology for connecting computers is available; many varieties of local-area networks are on the market, and most are fast and reliable.

However, the infrastructure which is necessary to manage and control distributed information has hardly been developed. The subject of our research at CWI is the design of such an infrastructure, a model that allows people to understand distributed computer systems and describe their actions.

Distributed computing is a new research area, one that introduces a whole range of new problems to be solved, problems of managing information systems without global and up-to-date information of their state, of finding ways to prevent inconsistencies in large bodies of data caused by unsynchronised simultaneous changes. Mechanisms must be found for protecting information against unauthorised access. The potential in distributed systems of much greater reliability must be used by designing services that can survive failures of individual components of the system. For some of these problems, solutions had already been found in traditional, centralised operating systems; other problems did not even exist before the advent of distributed computing.

Take the exploitation of parallelism, for instance: If there are two different programs to be run, two processors are evidently more powerful than one; the work can be divided. But this is not so evident if there is only one program to be run. It is then much harder to put the available parallelism to use. Traditional system design methods and software engineering principles do not provide adequate methods of splitting up algorithms in independent parts which can be executed in parallel. Building distributed systems is easy. Using them is hard.

Potentially, systems built up of many processors are more reliable than traditional computers with a single CPU. If the single processor of a centralised system fails, the system comes to a halt. In a distributed system, this does not have to be the case. Every single component of the system can be replicated, so that, no matter what component fails, a subsystem is left behind that can be made to work. If one processing element fails, others can take over the work. If a disk fails, a copy of the information can still be available on another disk.

As it turns out, designing software that exploits this fault-tolerant property of such a configuration is surprisingly difficult. Standard techniques for software development are all based on the assumption that the underlying hardware is infallible. This is a perfectly proper assumption in traditional systems, where, if part of the system fails, the whole system stops working, but it is no longer true in a distributed system.

Distributed systems research concentrates on the problem of structuring the hardware and designing the operating system software in such a way that we can profit from the architecture's two most important potentials, parallelism and fault tolerance.

Distributed control plays a central role in 'avoiding single points of failure.' Specialisation and control cannot be obtained through a simple hierarchical structure as exemplified by most armies. Again, the analogy with nature teaches us that extensive hierarchical systems can exist with a control structure

that provides enough redundancy to survive 'simple' failures.

The realisation of distributed control in all parts of the system is a key goal of the research: Any centralised part will be a potential bottleneck when the system grows, and a liability in the face of crashes. It is because of the importance of distributed control that we have named the distributed operating system emerging from our research ***Amoeba***, after that one-celled creature using distributed control to move about.

2. The Amoeba Distributed Operating System

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 a few hundred dollars in 1990. Our intention, therefore, has been 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-mapped 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 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' (as exemplified by Xerox Parc [5]) in a number of ways. In the personal computer model, each user has a dedicated minicomputer, complete with disks, in the 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 files. 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. This contrasts with the Cambridge Distributed Operating System [9], which also has a 'processor bank,' but assigns a processor to a user for the duration of a login session.

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 0.5 megabit/sec over a 10 megabit/sec local network, which is only 5% utilisation, 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*' [3] 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*' [6]. 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 convenient to *implement* the object model in terms of clients (users) who send messages to services [2, 9, 1]. 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 organised internally for both reliability and high performance, and so on. Analogous questions arise for all the other servers that comprise the basic system software.

3. Communication Primitives and Protocols

In the literature about computer networks, one finds much discussion of the ISO Reference Model for Open Systems Interconnection (OSI) [12] 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.

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. User programs execute

in the fourth layer, the Application Layer.

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. A *server process* makes a call to *getreq* (an abbreviation of *get request*) to tell the Transaction Layer it is ready to receive a request from a client. The client sends a request by calling *trans* (for *transaction*), which makes the Transaction Layer send a request and wait until a reply comes back from the server. The client is blocked until this reply arrives. The server, after carrying out the request, returns a reply by a call to *putrep*.

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 identity of 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.

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, compared to other approaches, our's leads to significantly higher transmission

speeds, due to simpler protocols.

Theoretically, very high speeds are achievable in modern local-area networks. A typical transfer rate of a modern local-area network is 500,000 bytes/sec point-to-point if there were no protocol overhead. In practice, 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. To do this, a large datagram size was chosen 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 obtain an efficient stream of packets.

Two versions of the algorithm have now been implemented. The one described above 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). A second implementation runs under UNIX,† using 2K byte datagrams, which 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.

4. PORTS

Every service has one or more *ports* [7] 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 authorisation, *etc.*

Communication using ports basically works as follows. One, or several server processes make a call to

getreq(*serverport*, · · ·);

a client, wishing to have some service rendered, calls

trans(*serverport*, · · ·);

The client's Transaction Layer generates a (unique) *reply port* for the client,

* PRONET is a trademark of Proteon Associates, Inc.

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

finds an active server process on *serverport* and sends a message containing

{*serverport*, *replyport*, · · · }

to the server. The server processes the request, and returns a reply in a message

{*replyport*, *serverport*, · · · }

The two ports provide a unique identification of the transaction.

The difference between this approach and that taken by conventional interprocess communication protocols is that, in principle, messages are addressed to a service name or *port*, not to a machine, or a process. Obviously, in Amoeba, clients never need to know where a service is implemented, or how many server processes there are. This is none of their business; it is part of the *implementation* of the 'abstract data type' that is the service.

The transaction mechanisms, however, must deliver requests and replies to specific processes on specific machines. Obviously, inside the Port Layer—which is, after all, responsible for message delivery—ports must be mapped onto networks addresses. So, when a client calls *trans*, the client's Port Layer must *find* a network node where a *getreq* on the matching port is outstanding.

In the local-area network, the technique for *locating* a port is the following. The client broadcasts a tiny message, saying '*anyone listening on port* x*?*', to which servers listening on that port reply '*port* x *is at machine* y*!*' A similar technique is used for locating the client for the reply message.

Locating ports is inefficient. It can be sped up, however, by a simple technique which finds many useful applications in distributed systems: the *hint*. In this particular case, hints are stored in a little table of (*port*, *network address*) pairs in every host and they say in effect: '*If you're looking for port* x, *why don't you try network address* y*?*' If the hint works, a port has been located without sending any extra messages; if not, a message is returned saying that the port is not known at that address. In the latter case, the hint is scratched out, and a broadcast locate is done. Incoming packets contain a source address, so they provide a free hint for their source port.

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 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 ensure 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 [2, 11].

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.

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 [13, 10, 4] 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 that the latter must have an inverse to be useful, but the former has been carefully chosen so that no inverse can be found.

Using the one-way F-box, the server authentication can be handled in a simple way, illustrated in FIGURE I. 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*). *Cap* is a *capability**, giving the identity of the object the user wants to access, which contains the *port* field, P, of the service managing the object. 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.

* Capabilities are explained in the next section.

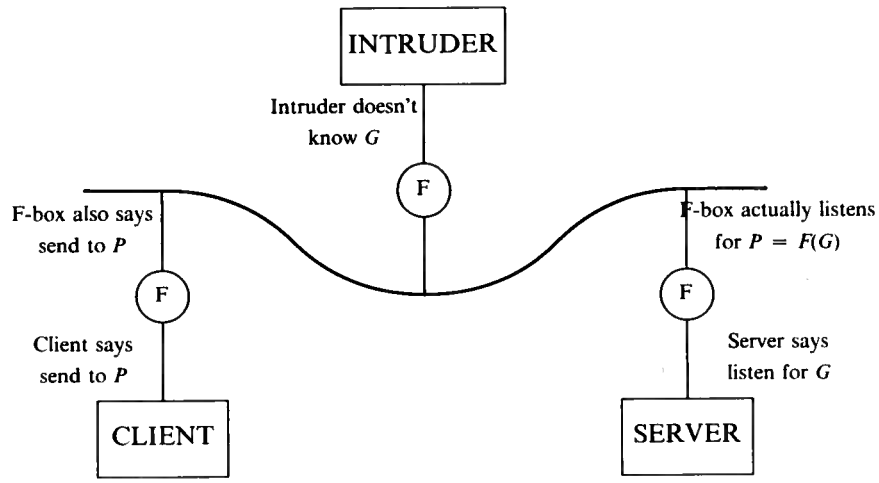

FIGURE 1.

Now let us consider the system from an intruder's point of view. To impersonate a server, the intruder must do *getreq*(G, $\cdots$). 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: The client's Transaction Layer picks a get-port for the reply, say, G', and the client's F-box transforms G' into $P' = F(G')$ in the request packet for the server to use as the put-port to send the reply to.

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 ensure 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.

5. 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 centralised 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.

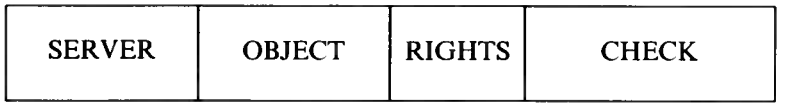

Figure 2.

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 Check Field 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 find 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 check field is computed by applying a one-way function to Object Number, Rights Field, and the Random Number stored with the object. When the capability is returned for use, the server uses the object number to find the file table and hence the random number. If the result of recomputing the Check Field leads to the Check Field in the capability, it 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 Check Field.

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 passing, say, read-only access, is harder. To accomplish this task, the process must send the capability back to the server with a bitmap saying which bits to strip off the Rights Field. By choosing the bit mask carefully, the capability owner can mask out any operations that the recipient is not permitted to carry out.

References

1. J. E. Ball, E. J. Burke, I. Gertner, K. A. Lantz, and R. F. Rashid, (1979). Perspectives on Message-Based Distributed Computing, *Proc. IEEE*.
2. D. R. Cheriton and W. Zwaenpoel, (October 1983). The Distributed V Kernel and its Performance for Diskless Workstations, *Operating Systems Review*, 17.5, 129-140.
3. J. B. Dennis and E. C. van Horn, (March 1966). Programming Semantics for Multiprogrammed Computations, *Comm. ACM*, 9.3, 143-155.
4. A. Evans, W. Kantrowitz, and E. Weiss, (August 1974). A User Authentication Scheme Not Requiring Secrecy in the Computer, *Comm. ACM*, 17.8, 437-442.
5. B. W. Lampson and R. F. Sproull, (1979). An Open Operating System For A Single User Machine, *Proc. Seventh Symp. on Oper. Syst. Prin.*, 98-105.
6. B. Liskov and S. Zilles, (April 1974). Programming With Abstract Data Types, *SIGPLAN Notices*, 9, 50-59.
7. S. J. Mullender and A. S. Tanenbaum, (1984). Protection and

Resource Control in Distributed Operating Systems, *Computer Networks*, 8.5,6, 421-432.

8. S. J. MULLENDER, (October 1985). *Principles of Distributed Operating System Design*: SMC, Amsterdam.
9. R. M. NEEDHAM AND A. J. HERBERT, (1982). *The Cambridge Distributed Computer System*: Addison-Wesley, Reading, Ma..
10. G. B. PURDY, (August 1974). A High Security Log-in Procedure, *Comm. ACM*, 17.8, 442-445.
11. R. RASHID (May 1981). Accent: A Network Operating System for SPICE/DSN, *Tech. Rept., Computer Science Dept.*, Carnegie-Mellon University.
12. A.S. TANENBAUM (1981). The ISO-OSI reference model, *IR-71*, Vrije Universiteit, Amsterdam.
13. M. V. WILKES, (1968). *Time-Sharing Computer Systems*: American Elsevier, New York.