Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

S.J. Mullender

A secure high-speed transaction protocol

Department of Computer Science Report CS-R8417 October

A Secure High-Speed Transaction Protocol

Sape J. Mullender

Centre for Mathematics & Computer Science
Amsterdam

Most computer networks use a byte stream protocol for communication between processes, which suffer from two important drawbacks: the addressing mechanisms provided are often process-dependent or location-dependent, and communication is slow. While carrying out research into distributed operating systems at the Vrije Universiteit and the Centre for Mathematics & Computer Science, we have developed a transaction-oriented transport protocol for the ***Amoeba*** distributed operating system [Tanenbaum81a], aimed for high-speed, with an addressing mechanism that is not only more general, but provides a protection mechanism as well. The basic mechanism for communication between processes is the transaction: a client process sends a request to a server process, which carries out the request and returns a reply. Protection is provided by using ports, chosen from a sparse address space, for addressing services. These ports serve as a "capability" for communication with the service. Through its simplicity, the transaction protocol achieves much higher transmission rates than other protocols executing on similar hardware (about 300 Kbytes/sec process-to-process).

The protection mechanism will be described, and the mechanisms for realising high transmission speeds.

1980 Mathematics Subject Classification: 68A05, 68B20. 69C22, 69C24 - 69D54
1982 CR Categories: C.2.2, C.2.4, D.4.4.
Keywords & Phrases: transaction protocols, connectionless protocols, capabilities, local-area networks.
Note: To appear in the Proceedings of the EUUG Conference 1984, Cambridge, UK

1. Introduction

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

Report CS—R8417
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

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." 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 inspect the representation of an abstract data type directly 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. Protection

Every service has one or more *ports* [Mullender82] 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 `put(port, message)` and `get(port, message)`. Since everyone knows the port of the file server, as an example, how does one insure

that malicious users do not execute `get`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 `get` 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 – if necessary – in 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 [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.

Using the one-way F-box, the server authentication can be handled in a simple way. Each server chooses a get-port, G, and computes the corresponding put-port, P. The get-port is kept secret; the put-port is distributed to potential clients or in the case of public servers, is published. When the server is ready to accept client requests, it does a `get(`G`)`. The F-box then computes $P = F(G)$ and waits for packets containing P to arrive. When one arrives, it is given to the process that did `get(`G`)`. To send a packet to the server, the client merely does `put(`P`)`, which sends a packet containing P in a header field to the server. The F-box on the sender's side does not perform any transformation on the P field of the outgoing packet.

Now let us consider the system from an intruder's point of view. To impersonate a server, the intruder must do `get(`G`)`. However, G is a well-kept secret, and is never transmitted on the network, Since we have assumed that G cannot be deduced from P (the one-way property of F) and that the intruder cannot circumvent the F-box, he cannot intercept packets not intended for him. Replies from the server to the client are protected the same way, only with the client picking a get-port for the reply, say, G', and including $P' = F(G')$ in the request 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 `get` has been done, the second is used as the put-port for the reply, and the third can be used to authenticate the sender, since only the true owner of the

signature will know what number to put in the third field to insure that the publicly-known $F(S)$ comes out.

It is important to note that the F-box arrangement merely provides a simple *mechanism* for implementing security and protection, but gives operating system designers considerable latitude for choosing various *policies*. The mechanism is sufficiently flexible and general that it should be possible to put it into hardware with precluding many as-yet-unthought-of operating systems to be designed in the future.

3. Communication Primitives

In the literature about computer networks, one finds much discussion of the ISO OSI reference model [Tanenbaum81b] 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.

3.1. 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 [Mullender]. 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 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.

3.2. 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 ports, the transport of (32K byte) datagrams (packets whose delivery is not guaranteed) from source to destination and enforces the protection mechanism of the previous 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

† UNIX is a Trademark of Bell Laboratories.

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.

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 containing the request is sent to the server and a timer is started. 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 (usually 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.

4. 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 example of a local-area network interface is shown in Fig. 1. When a host transmits a packet to another host, the packet is first transferred to the interface by means of a *direct memory access* (DMA) transfer. When this is done, the packet is transmitted over the network. After the packet has been received by the destination interface, it can be transferred to the destination host's memory, again using a DMA transfer. While this transfer is going on, the sending host may already transfer the next packet to the interface. A sequence of packets is thus transmitted by interchanging periods of DMA transfers and network transfers. On most interfaces DMA transfers and network transfers cannot occur simultaneously.

It is now simple to deduce an upper bound for the maximum transfer rate over the network: A typical speed for DMA transfers is 1 byte/μsec, and the typical

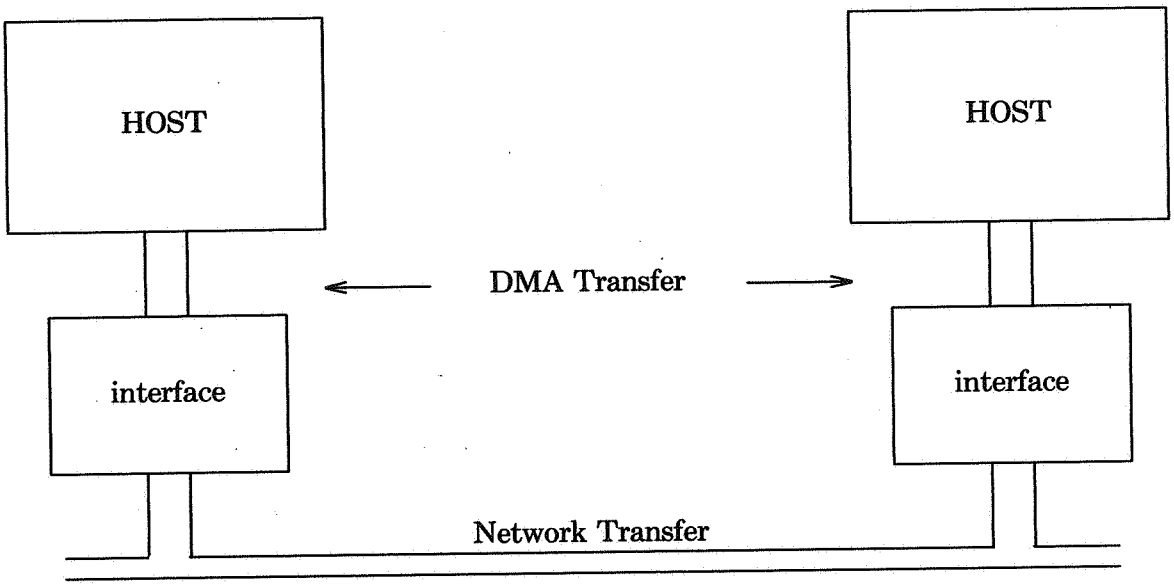

Fig.1. A typical local-area network interface.

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.

Obviously, to achieve such a transmission rate, the overhead of the protocol must be kept as low as possible, while an effort must be made to overlap DMAs at both communicating parties. To achieve this, we have chosen a very 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 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. We discovered that 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 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 protocol 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 it is 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.

5. 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;
```

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.

5.1. 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 called every n clock tics. N must be chosen such that n tics 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).

5.2. 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 Pro-net 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.

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.

The *port* mechanism allows us to move services from one machine to another, completely transparently to the user. The F-boxes do not yet exist in hardware, but are built into the operating system. The one-way function does not significantly slow the system down, because a cache is maintained of get-port/put-port pairs.

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.

[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.

[Mullender82]
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.

[Mullender]
Mullender, S. J., Renesse, R. van, and Tanenbaum, A. S., "A Transaction-Oriented Transaction Protocol," *In Preparation.*

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

[Tanenbaum81a]
Tanenbaum, A. S. and Mullender, S. J., "An Overview of the Amoeba Distributed Operating System," *Operating System Review.*, vol. 15, no. 3, pp.51-64, July 1981.

[Tanenbaum81b]
Tanenbaum, A.S., "The ISO-OSI reference model", IR-71, Vrije Universiteit, Amsterdam, 1981.

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