Protocols for Local Area Networks: A Survey

Nicolien J. Drost

RUU-CS-88-2
February 1988

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

Protocols for Local Area Networks: A Survey

Nicolien J. Drost

Technical Report RUU-CS-88-2
February 1988

Department of Computer Science
University of Utrecht
P.O.Box 80.012
3508 TA Utrecht
the Netherlands

Protocols for Local Area Networks: a Survey.

Nicolien Drost

February 1988

Abstract

This paper contains a discussion of possible applications of Local Area Networks, a brief survey of OSI protocols, some remarks on ways of improving the performance of LANs, and a more extensive treatment of Blast protocols and special protocols for Remote Procedure Call.

Contents

1 Introduction

More and more stand-alone computer systems are being replaced by or connected to computer networks. A computer network is an interconnected collection of autonomous computers [Tane81]. In more general terms: a network is a collection of nodes connected by links. Local Area Networks (LANs) are characterised in [Tane81] as having:
1) A diameter of not more than a few kilometers,

2) A total data rate exceeding 1 Mbit/sec,
3) Ownership by a single organisation.
Network software is responsible for the communication between the computers.

If computers wish to communicate, they need a protocol: a set of rules for the formats of the messages and for the process of exchange of messages. To get out of the Tower of Babylon problem protocols should be standardised. The most widely known standard for networks is the ISO-OSI Reference Model (OSI/RM84). The model consists of seven layers. Each layer provides services to higher layers. Entities in one layer communicate with each other by using services of lower layers. The tasks of the various layers are:

1. Physical Layer : bit transport,
2. Datalink layer : error detection, link control,
3. Network layer : routing, congestion control,
4. Transport layer : flow control,
5. Session layer : dialogue management, synchronization,
6. Presentation layer : conversion, datacompression, encryption,
7. Application layer : application-oriented services.

A communication service of a layer can be connectionless or connection-oriented. A connection-oriented service explicitly establishes and releases a connection. In this way reliable data transport without loss, duplicates and misordered messages can be realised, provided messages do not get delayed arbitrarily long. In a connectionless service the data is sent in independent datagrams, each containing the address. Data transport is now no more reliable if the communication channel may lose messages.

The OSI-reference model is a general model, and hence does not use the special properties of LANs optimally. LANs have communication channels with high throughput and low error rate. The seven layer structure of OSI causes a considerable overhead. Each layer adds a header to the message, and communicates with a lower layer.

There are three approaches to solving this problem: reducing the number of layers, using special protocols, and improving the implementation strategies and techniques for general layered communication architectures. One special protocol is the Blast protocol. This protocol is especially suitable for large data transfers. Special protocols are also used for Remote Procedure Call. This is a mechanism by which a program on one machine may call a procedure on another machine.

In this paper I will briefly survey the standard OSI and IEEE protocols (section 3), give some comments on strategies for improving performance (section 4), and give a more extensive treatment of Blast protocol (section 5) and Remote Procedure Call (section 6).

2 Types and applications of LANs

The most commonly used classification of LANs is based on the type of communication medium, topology, and datalink protocol used [Stal84a]. Physical media currently in use for LANs are twisted pair wire, coaxial cable (baseband and broadband) and optical fibre. Common topologies are the star, ring, tree, and bus. In the bus topology the medium is a

broadcast medium. A message from one computer never passes through another computer to reach ist destination. No more than one user may send a message at a time, otherwise collisions will occur. Only some combinations of topology and medium are possible:

	bus	tree	ring	star
Twisted Wire	x		x	x
Baseband Coax	x		x	
Broadband Coax	x	x		
Optical Fiber			x	

The types of datalink protocols now most frequently used are:

1. Carrier Sense Multiple Access/Collision Detection (CSMA/CD). These are protocols for buses. The sender listens until the cable is free, and only then sends a message. It continues listening to detect whether a collision occurs.

2. Token protocols. A token circulates on the network. Only a station that has the token is allowed to send. These protocols are used on rings and buses.

3. Slots. The topology is a ring. The sequence of bits circulating on the ring is divided into slots of fixed length. Slots are marked empty or full. A station may place a packet in an empty slot.

The existing combinations of medium, topology and protocol type are:

1. CSMA/CD bus (often called Ethernet),

2. Token bus,

3. Token ring,

4. Slotted ring.

Many existing LANS implement only layers 1 to 3 or 1 to 4. They only provide transport of messages from one host to another, not from one user process to another. Each host should contain a network server that sends messages from user processes on the network and delivers messages received from the network to the right user processes.

The application layer offers application-oriented services to end users. There are different ideas about which application should be implemented. The central question here is: for what purpose do we want to use a LAN? Most existing LANs are used for transport of files, mailing, and virtual terminal access. These are indeed the services of the application layer listed in the OSI reference manual [OSI/R84]. But new applications of LANs are developing. A LAN may be the basis of a distributed system.

There is considerable confusion in the literature about what a distributed system is. Some authors,e.g. Nelson [Nels81], use the name as a synonym for Local Area Network. It is also used to denote a system that permits distributed computing: different parts (or copies) of the same program run in parallel on different computers and communicate with each other. Distribution of program parts over the computers and control functions like termination detection, leader election and resynchronisation could be left to the programmer or delegated to a special operating system for distributed programming.

[Ensl78] requires a distributed system to have a systemwide operating system,with services requested by name, not by location. [TaRe85] call such an operating system a

distributed operating system. A distributed operating system may, but need not, support distributed computing.
[Hutc88] gives several models for distributed computing:

1. Hierarchical Model,
2. Client-Server Model,
3. Processor Pool Model.

In the Hierarchical Model a program sends subtasks to other machines to execute there. This is suitable for applications that have a hierarchical structure. Distributed databases are a special example of this. Many applications in industrial and commercial organisations also fall in this category.

This kind of system needs a service 'send a subtask to another machine' and message passing between program and subtasks. The first service does not fit very well into the OSI model: it is not viewed as a communication between two end users. Each machine should therefore contain a process, perhaps part of an operating system, that receives subtasks and starts their execution.

In the Client-Server model each program may request services from servers, usually situated on other machines. Examples of servers are: printer server, file server, special programs. Here the underlying LAN should provide a service 'request a given type of service'. This service could be implemented using Remote Procedure Call (see section 5), or a relaxed, asynchronous version of it.
Examples of Client-Server systems are :

- Cambridge Ring [NeHe81]. Each user has a simple terminal. If a user logs in, a Processor Server provides a processor from a processor pool.
- Xerox PARC Ethernet [Hutc88]. Each user has a desktop workstation and uses the network for shared resources.
- Grapevine[BLNS82]. This system ptovides message delivery, resource location, authentication, and access control services in a network.
- Amoeba[MuTa86]. This is a distributed operating system using Remote Procedure Call.

In the Processor Pool Model a program consists of several (possibly identical) parts, each running on a different machine. Possible programming languages are Concurrent Pascal and OCCAM. Here a programmer needs facilities to distribute program parts over nodes. Again this does not fit well into the OSI model as it is a priori no communication between processes. It is more reasonable to incorporate this service into a special operating system. Such a special operating system could also provide services like termination detection and resynchronization after crashes.

LAN-communication software could also provide services to a distributed operating system or a network operating system. [TaRe85] describe a network operating system as an extra layer superposed on existing (possibly different) operating systems in each node. Users mostly work on one machine and know on which machines their files are located. In a distributed operating system a copy of the operating system is present on each node

of the network. Users of the system do not know on which machine of the network their program is running and where their files are located. In such a system the boundary between layer 7 of the network and the operating system becomes rather vague. Services listed in the OSI reference model in layer 7 are here provided to user programs by the operating system, e.g. file transfer. It seems reasonable to include all communication tasks in the LAN-communication software and implement the operating system as user of the LAN.

3 Some OSI Protocols for LANs

3.1 Datalink Layer

The first standards for LAN protocols were the IEEE 802 standards. These belong to the physical and datalink layer in the OSI model. Later these protocols were also incorporated in the OSI model, with numbers 8802.2 to 8802.5. [Hals85] gives a description of these protocols:

IEEE 802.3 is a CSMA/CD protocol. If a collision occurs, the nodes use *truncated binary exponential backoff*: after the Nth collision a node waits a random number of intervals R with R drawn from a uniform distribution, $0 \leq R \leq K$, where K=min(N,Backoff Limit). Here loss of packets is possible if a new packet arrives before the receiver processed the previous packet.

IEEE 802.4 is a Token Ring Protocol. Every frame has a priority P_{frame}. Also the token has a priority P_{token}. If the token arrives at a node that wants to send a frame of a priority greater than or equal to the token priority, the node sends the token on with a new priority equal to the priority of this frame. If the token returns to the node with the same priority the node may send all frames with equal or higher priority as long as the Token Holding Time does not expire. Then the node sends the token again. The token gets back its old priority unless the node has still frames with higher priority to send. Then the token carries the highest of these priorities.

Every data frame carries a bit signifying if the receiver copied the data. So the sender is notified if a receiver did not accept the data.

One node acts as Active Monitor and checks regularly if there is still a token present. At intervals the Active Monitor sends an Active Monitor Present frame (AMP). Each node has an AMP timer. If it expires before an AMP frame arrived, the node sends a Claim Token(CT) frame. A node receiving a CT sends it on if the node is not the Active Monitor and did not send a CT itself, or did send a CT but has a lower node address than the one contained in this CT. If a node receives a CT with its own address, the node is the new Active Monitor.

IEEE 802.5 is a Token Bus Protocol. In error-free conditions, operation is similar to a Token Ring. To be able to cope with errors, every node knows the address of its successor and predecessor in the ring. If a node sends the token to its successor, it listens afterwards whether the successor reacts: sends a frame or sends the token on. If the node perceives no reaction, it broadcasts a 'Who Follows Me' message containing the address of its successor. Now the successor of the successor should react and become the new successor of the node, thereby excluding the inactive node.

To let nodes enter the ring again at random times, a Response Window procedure is executed. A node sends a Solicit Successor frame containing an address interval. If there

is a node wanting to enter the ring with its address in the interval, it answers. It is then entered into the ring. If a collision occurs (i.e. when more nodes want to enter the ring), the node repeats Solicit Successor with one address from the interval.

IEEE 802.2 contains two protocols: an unacked connectionless protocol, and a connection-oriented protocol. Connections are established and released by a 2-way handshake, and the sliding window protocol with go-back-N or selective retransmission is used for data transfer.

3.2 Transport Layer

The OSI Transport Protocol Specification[OSI/TPS84] contains five classes of transport protocols. They correspond to five classes of services. All protocols are connection-oriented. Connections are established (and later, released) in classes 1 to 4 by a 2-way handshake, in class 5 by a 3-way handshake. Data packets always have sequence numbers. Class 1 and 3 provide recovery from errors signalled by the network layer. Class 2 and 3 provide multiplexing. Class 4 uses a sliding window protocol to detect and recover from lost, duplicated, and out-of-sequence data packets. Each packet is acked. Timers are used for controlling retransmissions. The size of the window is also used for flow control.

3.3 Session Layer

For the Session Layer of the OSI-model one rather complicated model is defined [OSI/SPS84]. It has functions for connection establishment and release, production of a stream of activities using the connection, setting synchronisation points in an activity, and resynchronising an activity. Connections are established and released using a 2-way handshake. Activities may be started, interrupted, resumed, aborted, and ended. A session entity is only allowed to issue one of these commands if it has the token. It also needs the token if it wants to place a synchronisation point. All commands must be answered by an ack. Resynchronisation of an activity takes place by restarting the activity from some synchronisation point.

3.4 Application Layer

In the OSI model application protocols were developed for file transfer and management, job transfer and virtual terminal service. Also a set of services is being developed called Common Applicative Service Elements[OSI/CASE84, OSI/CASE85]. Some planned services are:

- Commitment, Concurrency, and Recovery.
- Reliable Bulk Transfer.

The CASE Basic Kernel Subset protocol is used for connection management. It uses a 2-way handshake. If an application entity receives a connect request is sends back a response containing 'accepted' or 'rejected'. A connection is closed in the same way.

4 Strategies to improve performance of LANs

LANs provide high speed transmissions and throughput and low error rate. Therefore an OSI- implementation for a LAN should contain protocols with little overhead as long as

no errors occur. Connectionless services and the corresponding protocols are in general simpler and have lower computational overhead than the connection-oriented counterparts, although they have a greater transmission overhead [Svob86]. So a datagram service is usually preferred.

Performance could also be improved by reducing the number of layers. This can be done by leaving out some layers. The topology of most LANs is so simple that routing and congestion control are not needed, and hence the network layer could be omitted. Not all LANs will need code conversion, data conversion, and encryption. So, in some cases, the presentation layer could be omitted.

The number of layers could also be reduced by combining different layers into one. This reduces the overhead caused by the different protocols and headers of each layer.

Special protocols are widely used in LANs to improve performance. [Svob86] mentions as examples the Blast protocol and special protocols for Remote Procedure Call. Blast protocols are treated more extensively in section 5 and Remote Procedure Call in section 6.

Many LAN-implementations use a combination of strategies to improve performance. An example is the Cambridge Ring [NeHe82]. It is based on a slotted ring with a raw data rate of 10 Mbits/sec. There are four levels of protocols: in level 1 the physical medium protocol; in level 2 a slotted protocol; in level 3 a connectionless protocol; and in level 4 protocols for rapid file transfer, high performance File Server protocols, and protocols for other applications.

5 Blast Protocols

The Blast Protocol is a special protocol devised for rapid transfer of large quantities of data over lines with few transmission errors. In a Blast protocol the sender transmits consecutive datapackets, each with a sequence number. Most Blast protocols establish a lightweight connection: the first data packet serves as an implicit connection request, and the last data packet carries an indication that it is indeed the last. The receiver only sends a message back after all data packets have been received or should have been received. The various possible contents of this message and of the reaction of the sender in case of an error give rise to different types of Blast protocols [Zwae85]:

1. Full retransmission without negative acknowledgement.
   The receiver sends an acknowledgement if all data packets arrived correctly, otherwise it sends nothing. If the sender receives no ack within a certain time after sending the last data packet, it retransmits all data.

2. Full retransmission with negative acknowledgement.
   The receiver sends a negative ack if not all datapackets arrived correctly. The sender retransmits all data on receipt of a negative ack.

3. Go-back-N.
   The receiver sends the sequence number of the first packet that arrived incorrectly or not at all. The sender retransmits all packets having a sequence number greater than or equal to the number received.

4. Selective retransmission.
   The receiver sends a bit map indicating which packets were received incorrectly or not at all. The sender retransmits only those packets.

An example of a Blast protocol is NETBLT [CLZ86]. NETBLT is a transport level protocol. It is implemented on top of the TCP/IP protocol [Stal84b]. NETBLT sets up a connection explicitly by exchanging two packets. The sending NETBLT-user provides a buffer with data. NETBLT breaks the buffer up into packets and sends the packets over the network. Typical packet sizes are 500-2000 bytes. The receiving NETBLT-entity loads these packets into a matching buffer provided by the receiving NETBLT-user. Theoretically buffers could be as big as 100K bytes. After the arrival of the last packet of the buffer, the receiving NETBLT-entity sends back a bit map for selective retransmission. When all packets have arrived correctly, the sending NETBLT-user provides a new buffer with data. This continues until all buffers have been sent. All buffers except possibly the last must be of the same size. Several buffers may also be transmitted concurrently, which improves performance markedly. NETBLT uses a novel kind of flow control called *rate control*. A sender sends a number of packets in quick succession (this is called a burst), and then waits a certain time before sending the next burst. Sender and receiver negotiate the size of the bursts and length of intervals between bursts during connection setup and after each buffer transmission.

The receiving NETBLT uses three timers. At the receipt of the first data packet of a buffer a data timer is set. If it expires the receiver assumes the whole buffer has been transmitted and acts accordingly. After sending a control message the receiver sets a control timer. Control messages have their own sequence numbers, independent from the sequence numbers of the data packets. The sender sends in each data packet also the sequence number of the last received control message. At the receipt of this number the receiver clears the timer. If a control timer expires, the receiver retransmits the control message. After a fixed number of retransmissions the receiver closes the connection. Both sender and receiver use a death timer that is reset every time a message is received. If a death timer expires, NETBLT assumes the other side has crashed and closes the connection.

Blast protocols are very fast for LANs because very few acks are sent. If the error rate of connections in the network is low, very few retransmissions will be needed. A disadvantage is the need for big buffers. [Reed82] describes a Blast protocol called BLAST in which packets are directly copied from one file to another. BLAST uses selective retransmission. Duplicate packets are no problem because they are copied to the same place in the file.

[Zwae85] compared Blast protocols to stop-and-wait and sliding window protocols. He analysed these protocols theoretically, and also did performance tests. He assumed:

1. Communication lines are half duplex,
2. The receiver always has enough buffer space to receive the data,
3. Sender and receiver work with approximately the same speed,
4. Transmission errors and interface errors are possible,
5. The network is never heavily loaded.

As a measure of efficiency Zwaenepoel used the total time needed for sending and receiving all data and acks. For the performance tests he used a 10 Mbit/sec Ethernet with Sun workstations. The protocols are part of the V kernel, a distributed operating system [Cher86].

The expected times for data transport are dependent on the number of data packets N, the time needed to copy a data or ack packet into or out of the network interface (C_{data} and C_{ack}), the time needed to put a data or ack packet on the communication line (T_{data} and T_{ack}, determined by the size of the packets and the speed of the line), and the latency r (the time needed to travel from source to destination).

[Zwae85] first derived formulas for the expected time for all three protocols. He used the simplest form of blast protocol: full retransmission without negative ack. In the Stop-and-Wait protocol the sender copies and transmits a data packet. Then the receiver copies the data and copies and transmits an ack. Only when the sender has copied the ack it may start to copy the next data packet. Hence the formula for the expected time needed to send and receive a blast of N packets is:

$$T_{StopAndWait} = N.(2C_{data} + T_{data} + 2C_{ack} + T_{ack})$$

In the sliding window protocol the sender does not have to wait for an ack before sending a new packet. But the interface cannot at the same time copy data from the sender and copy an ack to the sender. So the sender copies and sends N data packets and copies N acks. In the last round the sender copies an ack while the receiver copies a data packet and copies and transmits an ack. Hence the formula is:

$$T_{SlidingWindow} = N.(C_{data} + T_{data} + C_{ack}) + C_{data} + T_{ack}$$

In the Blast protocol the sender copies and transmits N data packets. After the sender has sent the last data packet, the receiver still has to copy this packet and copy and transmit an ack and the sender has to copy this ack. This gives:

$$T_{Blast} = N.(C_{data} + T_{data}) + C_{data} + 2C_{ack} + T_{ack}$$

Figure 1, taken from [Zwae85], pictures the transport of two packets using these protocols, and also for a blast protocol using two buffers.

[Zwae85] tested these protocols both in an otherwise unused network and in a fully operating one. He tested versions of the protocols without error recovery. The execution simply stopped when an error occurred. Data packet size was 1 K bytes, ack packet size was 64 bytes, and latency about 10 μsec. In a 10 Mbit/sec network this gives $T_{data} = 0.82$ msec and $T_{ack} = 0.05$ msec. The results were (in msec):

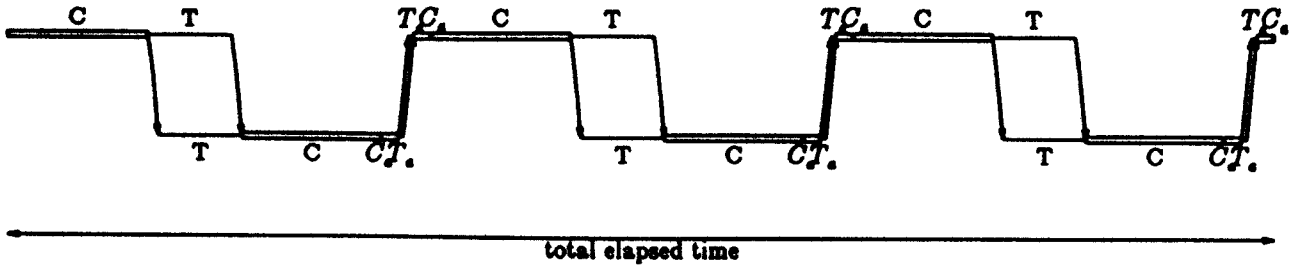
Empty network:

N	Stop And-Wait	Sliding Window	Blast
1	4.1	4.1	4.1
8	32.7	21.7	19.8
64	261.6	161.5	141.1
512	2093.0	1284.0	1149.0

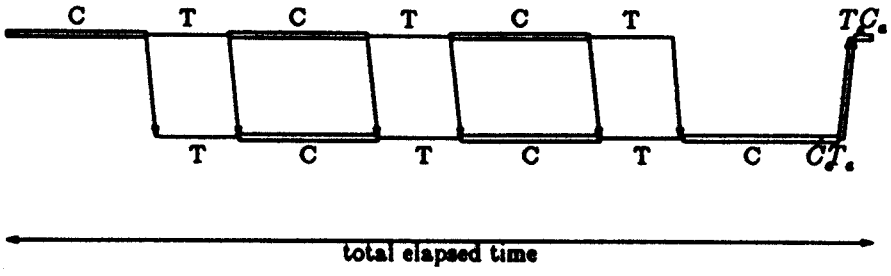
Fully operating network:

N	Stop And-Wait	Blast
1	5.9	5.9
8	43.1	24.8
64	340.2	73.0
512	2719.0	1370.0

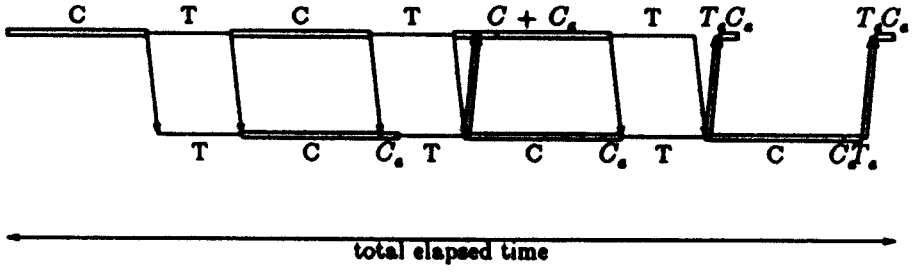
The stop-and-wait protocol takes 50-80 % more time than sliding window and blast. The sliding window protocol takes around 10 % more time than the blast protocol. From the test in the empty network it can be deduced that the times for copying into and out



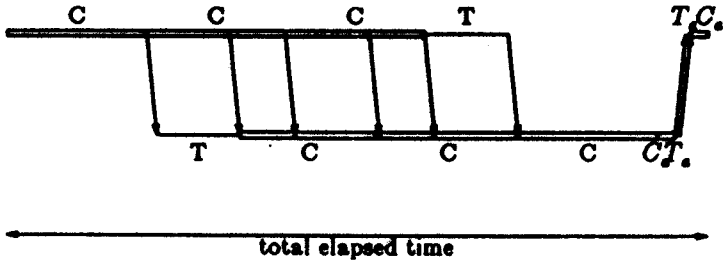
a: Stop-and-Wait



b: Blast Protocol



c: Sliding Window Protocol



d: Double Buffered Interface with Blast Protocol

Figure 1: Transmission of two data packets and corresponding acks using various protocols

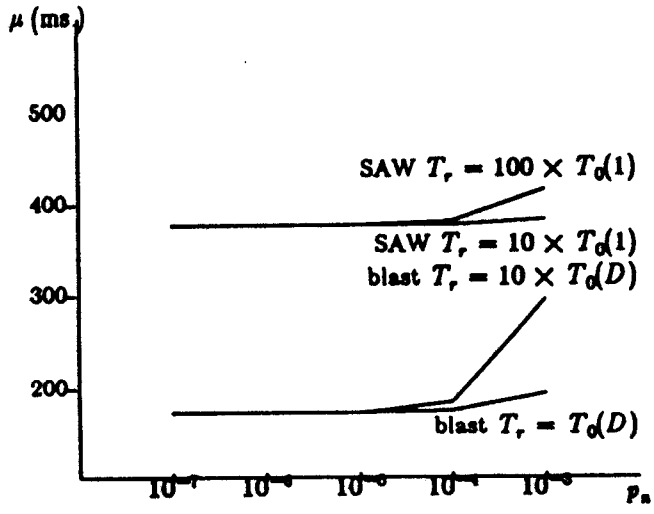


Figure 2: Expected time for datatransport as a function of error rate and timer value

of interfaces were: $C_{data} = 1.35$ msec and $C_{ack} = 0.17$ msec. So this takes more time than putting the data on the line ($T_{data} = 0.82$ msec, $T_{ack} = 0.05$ msec).It would be an advantage if copying into and out of the interface could be avoided. Most existing systems do not permit this. [Zwae85] did not test the computational overhead of the different protocols.

In his analysis of the behaviour of the protocols in the presence of transmission errors [Zwae85] only considered Stop-and-Wait and Blast, and did not test performance in a real system. Now the protocols set timers and retransmit if an ack does not arrive in time. The expected times are now dependent on the probability p_c that an error occurs during the transfer of a packet, and on the time T_r of the timer. The formulas for the expected times are:

$T_{StopAndWait} = N.(T_0(1) + (T_0(1) + T_r).p_c/(1 - p_c)$
$T_{Blast} = T_0(N) + (T_0(N) + T_r).p_c/(1 - p_c)$

where $T_0(1)$ and $T_0(N)$ are the time needed for a one-packet and N-packet exchange without errors using the same protocol. Figure 2, taken from [Zwae85], gives the expected time as a function of the error rate for different values of the timer. For low error rates Blast is significantly better than Stop-and-Wait. For higher error rates the timer value becomes very important. Too high a value causes long waiting times, but too low a value causes unnecessary retransmissions. The optimum value is dependent on the fluctuations in the time needed for the data transfer. If the network is always lightly loaded, as is the case in most LANs, these fluctuations will be small. In his network [Zwae85] measured an error rate of 1 error in 100.000 packets in normal circumstances, and 1 in 10.000 if one workstation transmitted at full speed to another workstation.

There is also an optimal value for the number of packets in a blast, depending on the error rate. If this number is too high, the number of retransmissions increases too much.

[Zwae85] also analysed the behaviour of more sophisticated forms of the blast protocol. The difference in performance between Blast with full retransmission without negative

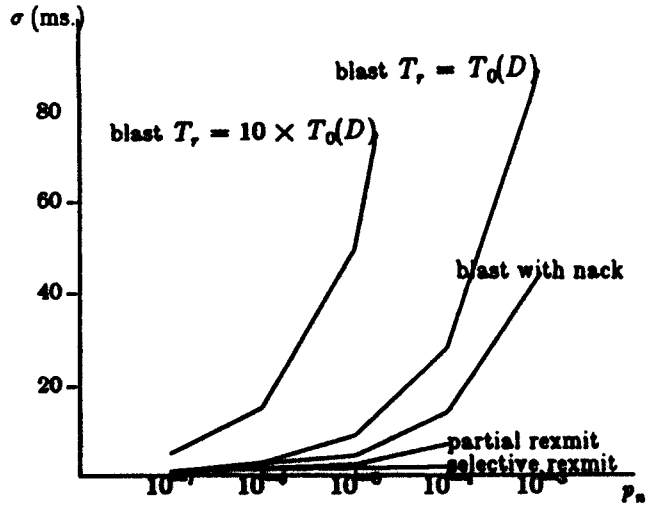


Figure 3: Standard Deviation of Data Transport Time for some Retransmission Strategies of Blast.

ack in circumstances without errors and with a low error rate are very small. A more sophisticated retransmission strategy cannot improve performance here, and would only introduce extra computational overhead. But [Zwae85] shows that different retransmission strategies influence the standard deviation of the expected time. His results are shown in Figure 3. This could be important for real-time applications. [Zwae85] therefore prefers a Blast protocol with go-back-N retransmission strategy.

6 Remote Procedure Call

Remote Procedure Call (RPC) is a mechanism by which a program in one node of a network can make a call to a procedure residing in another node of the network. [Nels81] defines RPC as follows:

> Remote Procedure Call is the synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel.

In a remote procedure call, the caller sends the parameter values to the node on which the called procedure (the callee) resides and suspends itself. The procedure is executed and the results, if any, are sent back to the caller, which resumes execution after receipt. RPC may be used for communication between programs [Nels81] or for requesting services from servers [Svob84]. The application layer of an OSI network could provide special protocols to support an RPC mechanism.

Local procedure call is a mechanism with a well understood semantics. Hence it would be convenient if RPC could have the same semantics as a local procedure call. But there are problems with RPC that do not occur in local procedure calls. Not all parameter types permitted for local procedure calls are possible for RPC. For example, some types could be represented and implemented in a different way on the caller and callee machine. As

the caller and the callee have no shared memory, no call-by-reference parameter passing is possible and extensiva data transports may be required to emulate it. Furthermore, authorisation and protection mechanisms may have to be added. Also it proves difficult to maintain the same call semantics for RPC and local procedure calls in the presence of node and line failures. Several call semantics have been proposed (cf. section 6.3) and a number of special protocols for RPC were devised to achieve these semantics.

6.1 Types and Parameters

If implementations of types are the same on all computers involved, value-parameters give no problems with RPC. If there are different implementations of one type, there should be a mechanism to translate implementations into each other. In a local call of a procedure with var parameters, the formal parameters are replaced by the addresses of the actual parameters. If there is no distributed operating system providing a global address space, these addresses are meaningless in other computers than the one where the variables reside. Even if there is, the operating system would need to send the value of the var parameter to the callee and send the new value of the var back to the caller. So a solution of the problem is to substitute call-by-reference by call-by value-result. The value of the var parameter is placed in the remote call, and the new value arriving with the results of the call is copied back into the variable.

Pointers offer harder problems. The value of a pointer is always an address in some address space, and meaningless in other address spaces. If a procedure has a pointer parameter pointing to some compound value that contains no other pointers, like a record, this pointer parameter should be substituted by a variable for this compound value. More difficult is a pointer parameter pointing to a structure containing other pointers, like lists and trees. Methods for transmission of graph stuctures exist, but are expensive in time and memory requirements [Nels81]. Shared structures give extra problems.

6.2 Binding

If a process makes a remote procedure call, it must be determined to which node of the network this call should be sent. A primitive way to solve this is to require a destination parameter in the call itself. This is not recommended because it makes the system very inflexible. Much better is *named binding*: in a call only the name, not the location of the procedure has to be given. The location could be obtained by table lookup, by using a name server, or by sending a broadcast to all nodes. The communication cost of the latter method is rather high, except when broadcasting a message to all nodes is as expensive as sending a message to one node. Therefore it is usually preferred to have in each node, or in a selected number of nodes, a list with names and locations. The CEDAR system [BiNe84] uses a database for this, copies of which are present on strategically placed nodes. RPC itself could also be the mechanism for obtaining location information, by implementing name server procedures on some nodes. Then each node needs to know at least one location of a name server, or must broadcast the location request. It is possible to place copies of procedures on different nodes.

6.3 Semantics of RPC in absence and presence of network failures

In the absence of node failures, local procedure calls have *exactly-once semantics.* If no transmission errors occur, a very simple protocol for RPC also guarantees exactly-once-semantics: the parameters obtained from the caller are sent to the callee, and the result (or an ack if there are no results to send) from the callee is sent to the caller.

If the RPC mechanism uses an unreliable data communication service, messages may get lost or delayed. To provide against loss of a call a timer could be set, with retransmission of the call if the timer expires before the result of the call arrived. But this can give rise to duplicate calls, if the first call was only delayed or took a long time to complete. To enable the caller to detect duplicates, the calls can be given sequence numbers. Also the caller may send a message 'call received' to the caller on receipt of the call. [Lamp81] describes some protocols for RPC using timers.

A message containing results of a call may also get lost. For this problem there are two solutions. The callee may store the results of the call, wait for an ack from the caller, and retransmit the results if this ack does not arrive within a certain time. If storing results is more expensive than re-running the call, then the callee may discard the results after sending them to the caller. The caller can then ask for a re-run of the call if the results did not arrive in time. In this case the RPC semantics are no longer necessarily exactly-once. This is no problem if the procedure is *idempotent*, i.e. when all executions with the same parameter values give the same results [Nels81]. Here *at-least-once semantics* are sufficient. An example of an idempotent procedure is a statistical test. There are also procedures that give wrong results if executed more than once, for example a procedure that places new values, calculated from the old ones, in a file. Here exactly-once semantics are needed. Finally, there may be procedures that give different results when called more than once with the same parameter values but with each result acceptable, like, for example, procedures that read values from a file that changes over time. In most cases the result of the last call is preferred, so *last-of-many semantics* are needed.

[Spec82] gives an overview of semantics that result if different protocols are used and various failure types may occur. He also describes a protocol that produces exactly-once semantics in case of failures. The caller sends the call to the callee, and repeats this call until a result comes back. Each call has a sequence number, which is used by the callee to detect duplicate calls. The callee stores the result of the call until it has received an ack from the caller. This ack contains the sequence number of the call, and also serves as an acknowledgement for calls with lower sequence numbers. If the callee receives a duplicate call, it sends again the result of the call to the caller. In this protocol it is essential that messages are delivered in the order they were sent, otherwise an ack from the caller could arrive before a duplicate call, and this could cause the callee to see this call as a new call.

Node crashes can give rise to orphans. An orphan is an outstanding call that continues to execute even though it was initiated by a caller on a now crashed node [Nels81]. Orphans are no problem if we want to achieve at-least-once semantics. For last-of-many semantics we need to know which results were produced by the last call. This can be reached by giving duplicate calls a version number. Also, orphans could be removed before the call is issued again. [Nels81] gives some techniques for this.

Exactly-once semantics is difficult to achieve in the presence of node crashes. Callers must store information in stable storage about the sequence numbers of their outstanding

calls, in order to be able to receive results after the crash and recovery of the caller node. A callee should store such information about the execution of the call that it is able to continue or restart the call after recovery from the crash. It depends on the properties of the procedure if this is possible.

6.4 Some implementations of RPC

Protocols for RPC could be part of the highest layer of the network software. If programs directly use services provided by the network software, a remote call consists of a call to the network service with as parameters the name of the remote procedure and the actual parameters for it. In this case local and remote procedure call have different formats. The implementation given in [CaCa84] could be considered an example of this, although they does not use special protocols for RPC. Another method is used by [BiNe84]. They implemented RPC in their CEDAR programming environment as a means for communication between programs. The network used consists of Dorado computers connected by a 3 Mbit/sec Ethernet. RPC is implemented using interface modules called *stubs*. When a user makes a remote call, it invokes a procedure in the user-stub. The user-stub makes a packet with parameters and destination and sends this using the RPC Runtime package to the server stub on the callee machine. The server stub makes a local call, collects the results, and sends them back to the user-stub in the caller machine. The user-stub returns the results to the user. In this case there is no difference for the programmer between local and remote procedure call.

The user-stub obtains the location of the called remote procedure from a database with copies in some strategically placed nodes in the network. The RPC Runtime package is responsible for sending and receiving i retransmissions, acks, packet routing and encryption. It uses protocols especially devised for RPC. It guarantees exactly-once semantics if the call returns and *zero-or-once* semantics if an exception is returned to the user. A simple protocol is used, in which the caller sends a call to the callee with a sequence number and sets a timer. If no results have come back when the timer expires, the caller retransmits the call. If the callee receives a retransmission of a call already received but not yet finished, it sends an explicit ack packet to the caller. A packet with a new call arriving at the callee from the same caller also serves as an ack for the results of the former call.

In a more complicated protocol, the caller sends probes to the callee while the call is executing. The callee should answer each call with an ack. This allows the caller to detect crashes of the callee or link failure, and to send in that case an exception to the user.

[CaCa84] describe a real-time system with RPC facilities. RPC is used here in application programs. The system is a distributed real-time computer control system for a 28 GeV Proton Synchroton Accelerator of CERN. The hardware is a LAN with star topology. End-to-end packet transmission takes 5 to 7 msec. RPC was implemented in programming languages (P^+, Pascal, NORD-PL) to enable application programmers to use the network in an easy way.

[CaCa84] divide implementation of an RPC facility into 3 parts:

1. Provision of an RPC protocol,
2. Provision of an interface for each programming language,
3. Provision of an initial set of remote procedures.

Their implementation of RPC is rather primitive. In the call, the remote computer must be identified explicitly by the programmer. An access routine named REM accepts a runtime call descriptor, makes a packet containing network address, requested service and parameters, and sends this via a datagram service of the network to the indicated machine. There the call is executed, and the result parameters are returned as another datagram to REM. No mechanism seems to be present that provides against lost packets or node or link failure. The time overhead for a null RPC is 20 msec, of which 15 msec are spent in network hardware and software, and the remainder in the interpreter. Each extra actual parameter gives an additional overhead of 4 msec.

During 1979-1983, 400 remote procedures and 150 application programs relying on RPC were implemented. The remote procedures were of four types:

1. General system routines,
2. Device driver routines,
3. Routines for specific groups of application programs,
4. Routines which are an integral part of a single distributed program.

In the system of Carpenter and Cailliau 5 RPCs are executed per second on the average. RPC has become a most fundamental and cost-effective tool for their application programmers. Most disadvantages they signal are caused by their primitive implementation: restrictions on parameter types, number, and size, restricted length of names and a too primitive network service without flow control. One disadvantage they mention is inherent to RPC: the blocking of the calling process. For this reason RPC is not the right mechanism for distributed programming applications where parts or copies of one program are supposed to run concurrently on different nodes of a network.

7 Conclusions and discussion

This paper gives a survey of some standard and special-purpose protocols for Local Area Networks. A widely used standard for networks is the ISO-OSI standard. This is a general architecture model and protocol standard, not adapted to the special properties of LANs like high speed and low error rate. The low-level protocols of OSI are well defined and widely used. There are a number of protocols in the datalink layer specially devised for LANs: CSMA/CD, Token ring, Token bus.

Using all seven layers of OSI in a LAN architecture causes a lot of overhead. In some cases a LAN does not need the services of all layers. Then one or more layers could be omitted. Also more layers could be combined into one. As the use of line capacity and the error rate in LANs are usually low, a connectionless protocol in the transport layer suffices. An alternative is a Blast protocol. This protocol is particularly suited for large data transfers. There are several versions of Blast protocols. Most versions use a lightweight connection, a type of connection in between connectionless and connection-oriented. The protocol corrects detected errors and duplicates. [Zwae85] compared Blast protocols to Stop-and-Wait and sliding window protocols. He measured total time needed for data transport in a LAN with half duplex links. The Blast protocol proved the most efficient protocol of the three. The sliding window took around 10% more time, and Stop-And-Wait 50-80%. Most time (around 60%) was spent copying data into and out of interfaces.

If this could be avoided, a considerable increase of efficiency could be reached. For higher error rates the value of the timer becomes increasingly important: too high a value causes unnecessarily long waiting, too low a value causes unnecessary retransmissions.

In the presence of low error rates a more sophisticated form of error correction strategy gives no better performance than the simplest form: full retransmission without negative ack. But selective retransmission causes the lowest variance in transmission times, so this strategy is preferable for realtime systems.

[Zwae85] did not take into account computational overhead of the protocols. This should be further analysed. I expect the sliding window protocol to have a greater overhead than the Blast protocol.

Protocols for higher OSI layers are still being developed. Many OSI-implementations only contain layers 1 to 3 or 4. What services the application layer in an OSI-LAN should provide is determined by the requirements of the users of the communication services of the LAN. Standards exist for file transfer, mailing, and remote terminal access. The communication services could be used in a loosely coupled system, where users mostly work on one system, and only occasionally use another computer. Then file transfer, mailing, and remote terminal access could suffice. Many existing LANs are of this type. For a distributed operating system or a system that supports distributed computing, other services are needed. An example is "request an action of a server", with servers possibly residing on another node. This could be implemented using Remote Procedure Call. RPC could also be used for the hierarchical variant of distributed computing. If full parallellism is needed RPC is less suitable, because the calling process is blocked during the call.

RPC should be part of a programming language, used to write an operating system or used by programmers of a system that supports distributed computing. The LAN communication software could provide special protocols for RPC. If implementations of types differ in the nodes, there should be a mechanism to translate them into each other. In the OSI standard this is a service of the presentation layer. Also RPC requires transport of parameters.

Several semantics are possible for RPC in the presence of failures: exactly-once, at-least-once, zero-or-once, last-of-many. These need different, special protocols in which timers or sequence numbers are used. A crash of a process or a node may give rise to orphans: outstanding calls that continue to execute while the process that initiated the call is no longer present. For last-of-many semantics a mechanism is needed that removes orphans or discards all results except the last one.

A number of implementations of RPC exist. In simple systems a local call and a remote call have different formats. In more sophisticated systems they have the same format.

Experiences with RPC as a tool for application programmers are good.

As yet no standards exist for protocols for RPC. Services supporting RPC could well be a part of the OSI-application layer in the future.

References

[BLNS82] Birrell,A.D., R.Levin, R.M.Needham, and M.D.Schroeder: Grapevine, an Exercise in Distributed Computing. CACM 25,4: 260-274. 1982.

[BiNe84] Birrell,A.D. and B.J.Nelson: Implementing Remote Procedure Calls. ACM Trans. on Comp. Syst. 2,1:39-59. Febr.1984.

[CaCa84] Carpenter,B.E. and R. Cailliau: Experiments with RPC's in a Real-Time Control System. Software Practice and Experience 14,9:901-907. Sept. 1984.

[Cher86] Cheriton,D.: Request-Response and Interprocess Communication in the V Kernel. In: Networking in Open Systems. Lecture Notes in Computer Science 248. pp176-192. 1986.

[CLZ86] Clark,D.D., M.L.Lambert and L.Zhang: NETBLT: A Bulk Data Transfer Protocol. Network Working Group, RFC No 969, Lab. for Computer Science, MIT, Cambridge, Mass., Jan. 1986.

[Ensl78] Enslow,P.H.Jr.: What is a 'Distributed' Data processing System? Computer 11:13-21. Jan. 1978.

[Hals85] Halsall,F.: Introduction to Data Communications and Computer Networks. Electronic Systems Engineering Series, Addison-Wesley Publ. Comp, London, 1985.

[Hutc88] Hutchison,D.: Local Area Network Architectures. Addison-Wesley Publ. Comp, London, 1988.

[Lamp81] Lampson,B.W.: Remote Procedure Calls. In: Distributed Systems: Architecture and Implementation. Lecture Notes in Computer Science 105, pp.365-370. Springer Verlag, Berlin, 1981.

[MuTa86] Mullender,S.J. and A.S.Tanenbaum: The Design of a Capability-Based Distributed Operating System. The Computer Journal 29,4:289-300. March 1986.

[NeHe82] Needham,R.M. and A.J.Herbert: The Cambridge Distributed Computer System. Addison-Wesley Publ. Comp. London, 1982.

[Nels81] Nelson, B. J.: Remote Procedure Call. Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, USA. 201pp. 1981. Reprinted as Techn. Rep. CSL-81-9, Xerox PARC, Palo Alto, Calif.

[OSI/CASE84] Informations Processing Systems - Open Systems Interconnection - Specification of protocols for Common Application Service Elements. Part 1: Introduction. Draft Proposal, ISO/DP 8650/1, 1984.

[OSI/CASE85] Informations Processing Systems - Open Systems Interconnection - Specification of protocols for Common Application Service Elements. Part 2a: Basic Kernel Subset. ISO/DP 8650/2, 1985.

[OSI/RM84] Informations Processing Systems - Open Systems Interconnection - Basic Reference Model. International Organization for Standardization, International Standard 7498, 1984.

[OSI/SPS84] Informations Processing Systems - Open Systems Interconnection - Basic Connection Oriented Session Protocol Specification. ISO/DIS 8327, 1984.

[OSI/TPS84] Informations Processing Systems - Open Systems Interconnection - Connection Oriented Transport Protocol Specification. ISO/8073. ISO/TC97/06N3240, 1984.

[Reed82] Reed,D.P.: Computer System Structures. Annual Report 1981-1982, Lab. for Computer Science, MIT, Cambridge, Mass., 1982.

[Spec82] Spector,A.Z.:Performing Remote Operations Efficiently on a Local Computer Network. Comm. ACM 25,4:246-260. 1982.

[Stal84a] Stallings, W.: Local Networks. ACM Computing Surveys 16,1:2-41. March 1984.

[Stal84b] tallings, W.: A Primer: Understanding transport protocols. Data Communications: November 1984.

[Svob84] Svobodova, L.: File servers for Network-Based Distributed Systems. ACM Computer Surveys 16,4: 353-398. 1984.

[Svob86] Svobodova, L.: Communications support for distributed Processing: Design and Implementation Issues. In: Networking in Open Systems. Lecture Notes in Computer Science 248, pp176-192. 1986.

[Tane81] Tanenbaum, A.S.: Computer Networks. Prentice Hall Inc., London, 1981.

[TaRe85] Tanenbaum, A.S., and R. van Renesse: Distributed Operating Systems. ACM Computer Surveys 17,4:419-470. Dec.1985.

[Zwae85] Zwaenepoel, W.: Protocols for large Data Transfers over Local Networks. Proc. 9th Data Communications Symposium, Whistler Mountain, British Columbia, pp. 22-32. Sept. 1985.