INTEGRATING LOCAL AREA NETWORKS TO IMPROVE RELIABILITY AND PERFORMANCE

By

KENNETH CHI-KIN CHAN

B.Sc., University of British Columbia, 1984

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October 1986

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date Oct. 16, 1986

Abstract

A hybrid network comprising an Ethernet and a Cambridge Ring has been proposed by researchers in the Department of Computer Science at the University of British Columbia as a means for improving reliablility and performance of computer communication in a local area network environment. This thesis explores the practicability of this concept and constructs a generalization of this model, where an arbitrary number of LANs, of arbitrary technologies, may be used together in an integrated fashion. The goal is a set of software primitives which provides a connection-oriented message-based IPC interface, and allows a user to utilize multiple networks with relative ease. A number of relevent design issues, including host reachability, path selection, and load monitoring are dealt with in detail. Also discussed is an implementation of this software interface here at the University of British Columbia, developed on Sun workstations running 4.2 BSD Unix which are inter-connected by an Ethernet and a Cambridge Ring. Measurement results on the performance of the implemented software are included.

Contents

List of Figures

List of Tables

Acknowledgement

I would like to thank my supervisor Dr. Sam Chanson for his guidance and cooperation. His counsel is greatly appreciated. Thanks must go also to Dr. Son Vuong, who served as the second reader for this thesis. Lastly, I am grateful for the help of Rick Sample, who overcame a great number of difficulties relating to the computing facilities here on my behalf.

Chapter 1

Introduction

This thesis explores the complexities involved with interconnecting computers in a local environment with more than one local area network (LAN). The networks may be all of the same type (e.g. two Ethernets[1]), or they may be of different types (e.g. an Ethernet and a Cambridge Ring); the problem domain is not restricted to one or the other. However, for reasons which will be explained shortly, this thesis will concentrate on the latter case. The ultimate goal is an integrated network environment, where the existence of multiple supporting networks is transparent to the end user, and the problems of dealing with multiple networks are simplified for systems and applications programmers. It is essentially an expansion of the hybrid LAN idea first proposed in [Vuon83]. Before discussing the design issues and describing a software implementation here at the University of British Columbia, this chapter will first present the motivations for researching this topic.

[1]Ethernet is a trademark of Xerox Corporation

1.1 Motivations

A local area network is an interconnection of computers which are distributed over a small geographical area, usually no more than a few square kilometers. LANs have been steadily gaining popularity, as their characteristically high data rates and relatively low installation costs are well suited to small organizations which wish to better utilize their computing resources.

However, as there is no single type of LAN which is optimal for all possible applications [Chan83,Limb84], each installation has had to choose one which is most suitable to its particular needs. A local area network is characterized by its topology, access method, and an associated family of protocols. These protocols usually correspond to the lower 3 layers of the ISO reference model for Open Systems Interconnection [ISO81]. Sometimes layer 4, the Transport Layer, is also included. Some background on common LAN topologies and access methods are given in the following subsection. Experienced readers may skip to Subsection 1.1.2.

1.1.1 Background

The topology of a network refers to the configuration of interconnection between machines. For local area networks, the typical topologies are bus, ring, and star. A bus

network, as shown in Figure 1.1, consists of a single long line which is tapped into by the nodes. The connection is made in a passive way; that is, each node can listen to the signal on the line without disturbing it. A ring network consists of a series of point-to-point links which form a closed loop. Here the signal must pass through each ring interface unit, which also act as repeaters for the signal. Thus the nodes are considered to be active. An illustration of the ring topology appears in Figure 1.2. A star network has all of its nodes connected to a central hub, as depicted in Figure 1.3. All traffic between nodes must pass through this central hub. A star-shaped ring is also possible [Neil85]. More on LAN topolgies may be found in [Tsao84]. This thesis will concentrate mostly on bus and ring networks, since they are the most popular.

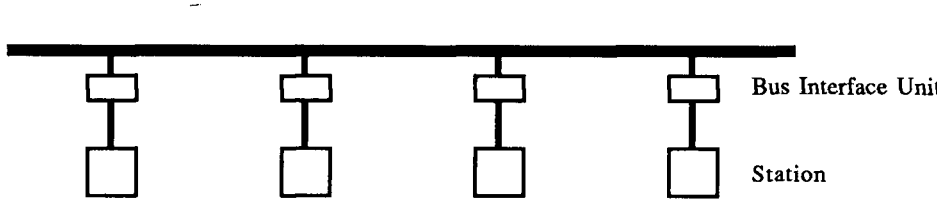


Figure 1.1: Bus Network

The transmission medium in local area networks is shared by all the nodes. Therefore some scheme is required to arbitrate the usage of the shared medium. This is referred to as the access method. For bus networks, a common access method is Car-

rier Sense Multiple Access with Collision Detection (CSMA/CD). In this method a node which wishes to transmit data will first listen to the medium and wait until it is idle before transmitting. After the transmission has begun, it must also check for collisions with another node which is also transmitting at that time. Collisions usually occur due to the propagation delay between nodes. A collision requires re-transmission by each contending node, after some random backoff time. Collisions may also be resolved deterministically, as is done in some variations of CSMA/CD [Neil85].

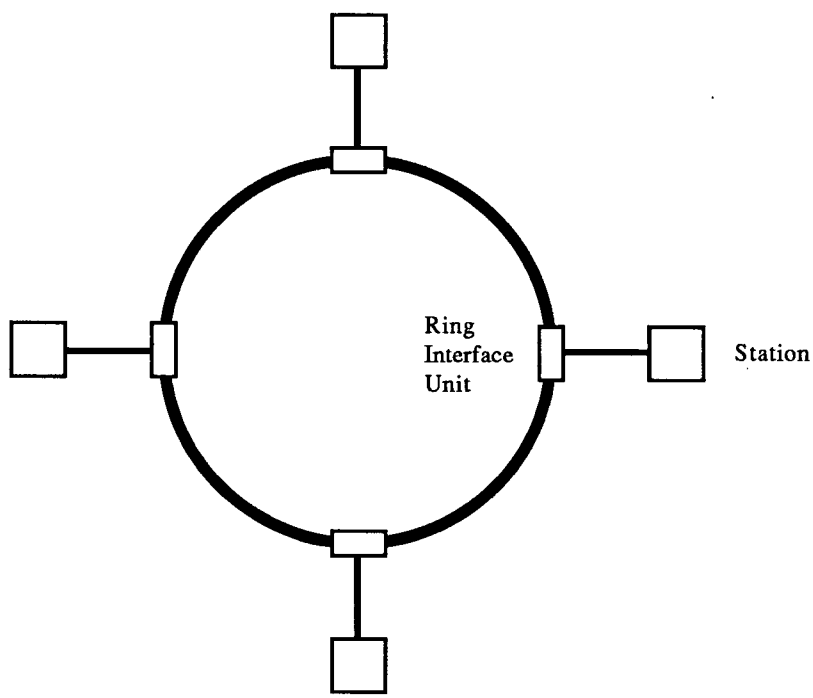


Figure 1.2: Ring Network

For ring networks, one scheme is to circulate a number of fixed-length slots contin-

uously around the ring. Each slot, when empty, may be filled with a fixed amount of data as it passes a station which wishes to tranmit. The station will then mark the slot as full, and re-mark it as empty after it has traversed the ring and returned. A variation of the slotted ring is the register-insertion ring. Here each node contains a shift register which can be used to buffer incoming data while the node simultaneously transmits outgoing data.

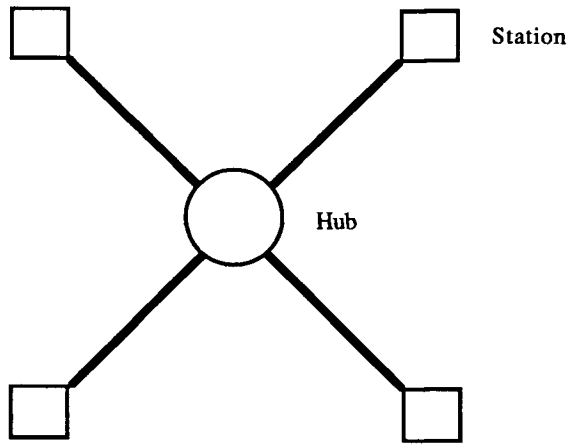


Figure 1.3: Star Network

One other access method, which is suitable for both buses and rings, is to circulate a unique bit sequence which acts as a token. A node must get hold of the token before it can transmit. It then destroys the token, regenerating it only after the transmitted packet has returned to it. In a token bus the token is passed from node to node in a

logical ring.

One of the most popular types of local area network is the Ethernet [Metc76], which is a CSMA/CD bus. Ethernet is often accessed through Arpanet's Transmission Control Protocol and Internet Protocol (generally referred to collectively as TCP/IP). Although TCP/IP [DARP81a,DARP81b] was designed for long haul networks (LHN), it is often used in LANs because it comes as part of the 4.2 BSD Unix[2] system. It is unclear whether the term "Ethernet" encapsulates the supporting protocols as well as the underlying topology and access method, or is simply the best-known example of (and hence could be a synonym for) a CSMA/CD bus. This thesis will use it in the latter sense. Ethernet has been standardized as a LAN technology by IEEE (802.3).

Another well-known type of local area network, particularly in Europe and the UK, is the Cambridge Ring [Wilk79], which is a slotted ring. The Cambridge Ring is usually associated with the Basic Block protocol (BB) and Byte Stream Protocol (BSP) [Dall81]. Like any other ring network, the Cambridge Ring suffers minimal signal attenuation over long distances. This is due to the fact that the signal is re-generated at each node. Because of the Cambridge Ring's suitability for covering a greater geographical area than most other LANs, it has been standardized by IEEE as a community area network technology rather than a local area network technology. Nonetheless, the Cambridge Ring is generally considered to be a type of LAN.

[2]Unix is a trademark of AT&T Bell Labs

Two other well-known LAN technologies are the token bus and the token ring. One token bus based network which is rapidly gaining recognition in industry is MAP (Manufacturing Automation Protocols). Developed by General Motors [GM85], MAP is intended for use in factories. The IBM Zurich Token Ring is an example of a network which is based on a token ring. IEEE has standardized the token ring as a LAN technology (802.5), as well as the token bus (802.4).

1.1.2 Problems

Unfortunately, each type of LAN is suited for a set of requirements which often conflict with the requirements of another type of LAN. For instance, because of its high data rate and variable packet size, Ethernet is well-suited for bulk data transfers, such as remote disk accesses. Its non-deterministic access method, though, means that it is unsuitable for real-time applications. The Cambridge Ring, on the other hand, provides a guaranteed worst-case access time, but is not as efficient for large volume data due to its small packet size (only 2 bytes of actual data per mini-packet) and hence lower data rate. LANs based on the other two technologies mentioned above similarly have their individual suitabilities and weaknesses for different applications. This implies that the selection of any single type of network will either exclude certain applications, or force those applications to run in an inefficient communications environment.

Recently, the emphasis has increasingly been placed on providing integrated services. There are certainly many situations where the ability to access a large variety of resources from a single source is advantageous. For example, a process engineer may wish to be able to monitor plant operations as well as utilize spreadsheet software and access an MIS database. Clearly conflicting requirements are involved here. The first is a real-time application which demands an environment such as MAP, while the latter two are best served by a network such as TOP (Technical and Office Protocols), an Ethernet-based network for office and technical applications [Boei85].

One approach to this problem is to connect some of the machines on one type of network, and to connect the rest of the machines on another type of network. The two networks are then bridged by a gateway node, as shown in Figure 1.4, which is connected to both networks. Machines on network A will generally run applications which are suitable for network A, and machines on network B will generally run applications which are suitable for network B. Machines located on different networks can still communicate, by going through the gateway node G. This partially solves the problem, as each type of application can usually be run in an appropriate environment, but are accessible from all stations. However, the gateway node presents a point of vulnerability where the two networks can become divided, as well as being a potential bottleneck. Also, an internetwork path will involve both types of network, and therefore it will suffer the combined disadvantages of both.

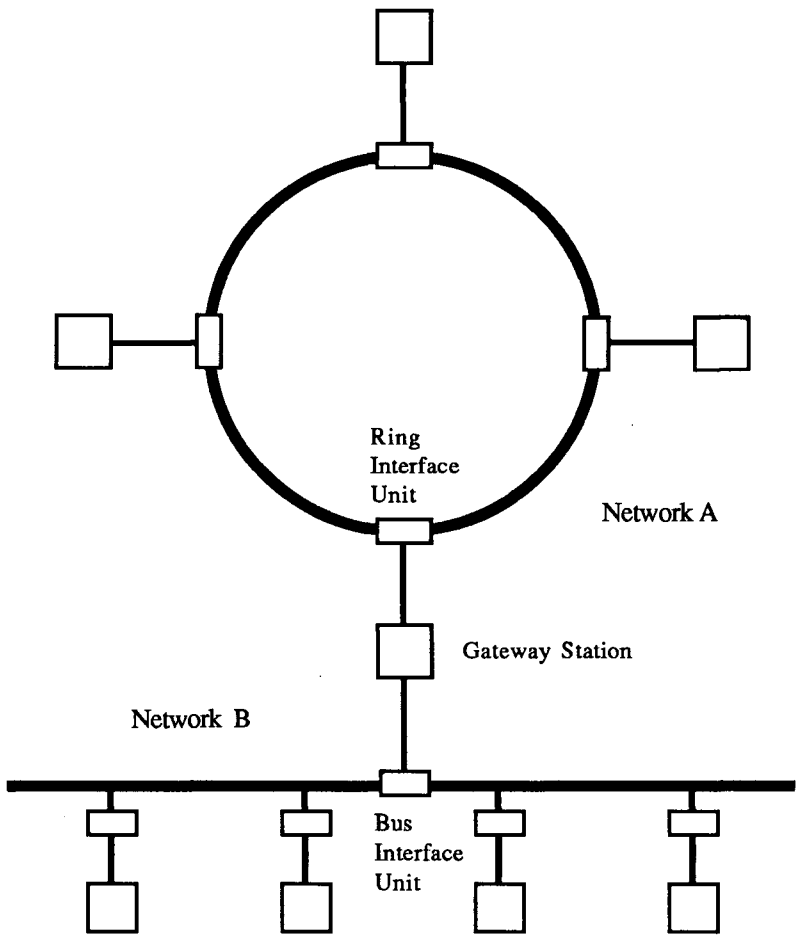


Figure 1.4: Two LANs Connected by a Gateway

As LAN technologies become more readily available and lower in cost, another feasible solution is to connect the machines with more than one network, as shown in Figure 1.5. This would allow each station to run each different type of application on the network which it is most suited for. In addition, reliability would be enhanced, since the unavailability of one network would not necessarily partition the machines. Furthermore, performance may be improved by allowing an application to use both networks simultaneously, or to select the network which is least congested.

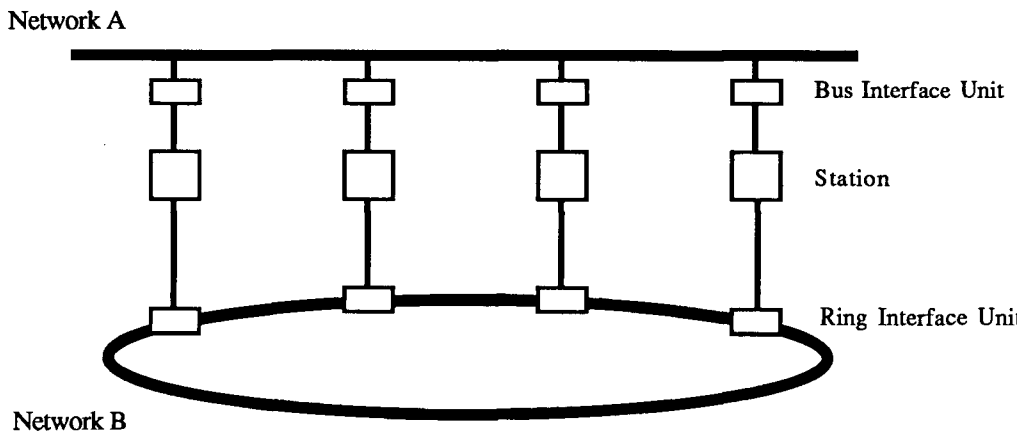


Figure 1.5: Stations Connected by Two LANs

One may argue here that LANs are usually highly reliable already, so that added reliability is not much of a gain. However, LANs may not be as reliable as people tend to believe. Results [Zwae85] have shown the error rates of the raw Ethernet transmission

medium to be on the order of 10^{-4}, rather than 10^{-6} as previously thought. This implies that LANs are not inherently reliable, and that they require reliable protocols. In addition, there exist applications where extra reliability is desirable, for instance military networks or process control involving hazardous substances.

One could also argue that currently processors are in general much slower than the transmission media, so that sending data over multiple paths simultaneously would not improve performance. Nonetheless, even if an individual station can use up no more than a few percent of a network's bandwidth, connecting hundreds of stations should produce enough traffic to significantly congest the network. Such large-scale LANs are becoming more common. Also, applications such as interactive graphics may require large volumes of data to be transferred in short periods of time. Multiple networks can also be viewed, therefore, as a way of attaining the extra bandwidth needed in order to interconnect a greater number of computers together directly without suffering seriously degraded performance.

1.2 Goals

How should the availability of multiple networks be made accessible to users? Since the end user's only concern is to run applications, the existence of multiple networks should be made transparent to him. To the systems or applications programmers,

though, awareness of multiple networks is useful, since each programmer would know his particular application's requirements and therefore the network which he would prefer to use. Thus the aim was to design a set of system primitives which allow user programs to utilize the networks available as they see fit, or to let the system make that decision if they do not care. Although there exist systems which provide access to multiple networks (for example, 4.2 BSD Unix allows a user to imply a particular network by specifying a protocol name), they do not provide these services in an integrated way. That is, the individual networks are viewed as being completely separate, so that if a user wants to use several network links together to achieve a single purpose, he himself would have to deal with problems such as loss of data or a link, which host is reachable on which network, and so on. Our view is that users should not have to deal with such problems; these should be handled by the system, within our set of primitives. In addition, these primitives should be easy to use so that existing applications can be easily converted.

To examine the practicality of such a service, another goal was to implement this set of primitives, so that measurements could be made to determine the overhead of this additional software. Moreover, it would allow us to see how easy or difficult it is to convert existing applications to this interface.

Before continuing further, the reader may be wondering why this work is concerned exclusively with local area networks, and not at all with long haul networks. Since the

alleged benefits of having multiple networks connected are reliability and performance, LHNs would seem to be prime candidates since they are much less reliable and have much lower data rates than LANs? However, the way which multiple LANs improve reliability and performance is by providing more than one path between machines, something which is already present in most LHNs due to their topologies. Therefore, in a sense what we are doing is applying techniques already being used in long haul networks to local area networks. However, we are generalizing these ideas, in that the access method and protocols are identical for all paths in an LHN, but the access method and protocols may be different for each path in our environment.

1.3 Thesis Summary

The preceding section has presented the motivations for working towards an integrated network, as well as the goals which we wish to achieve. Chapter 2 discusses in some detail the design issues which need to be dealt with, and some possible solutions. The approach was to work towards a general solution, for arbitrary types of LANs, rather than a specific solution which is geared to our particular implementation environment.

An implementation of the objective software module of system primitives, referred from here on as MIN (for "multiple integrated networks"), was realized here at U.B.C. Chapter 3 describes this implementation, in particular with regard to how each of the

issues discussed in Chapter 2 were dealt with. It also reports on the implementation environment, in terms of both the hardware and the supporting network software.

Chapter 4 gives the measurements results on the processing overhead incurred by adding our software. This was done by comparing the (real) execution times required to transfer a large amount of data over each network, with and without going through our software interface.

Chapter 5 looks back at what we have done and draws some conclusions on whether the effort has been worthwhile. It also suggests some possible enhancements and future work.

Chapter 2

Design

As mentioned in Section 1.2, our objective is a set of software primitives, which we've termed MIN for "multiple integrated networks". This chapter discusses the design of MIN. The approach is to derive a general model, for arbitrary local area networks and hosts, rather than a specific solution for our particular implementation environment. The discussion begins with the overall structure, in terms of the relationship between our software, the user, and the supporting network protocols. Next, the type of communication services that the MIN primitives should provide are described. A number of design issues, including host reachability, connection establishment, path selection, and load monitoring are also examined. The final section deals with the internal protocol employed by the MIN primitives for message exchange.

2.1 Overall Structure

The OSI reference model [ISO81] is widely accepted as a framework for communications design. Therefore it is advisable for us to begin by determining where within the 7 layers of the OSI model our objective software should reside.

With more than one network available, part of MIN's function is to choose one of the networks when data is to be transmitted. Making this choice in the Physical layer would not make much sense. In many cases the physical layer consists of a driver for the hardware, so that a separate one for each transmission medium is necessary. Placing MIN in the Data Link layer to make the routing decision there would correspond to the suggestion made in [Vuon83]. This scheme is illustrated in Figure 2.1. Note that the Presentation and Session layers are shown with dashed outlines because currently most systems do not actually contain these layers.

If the decision is made in the Link layer, then the networks must be logically integrated in the higher layers. That is, a uniform protocol must be designed for the Network and Transport layers which could be used for all networks. There are a number of difficulties with this idea. Firstly, the functions provided by each layer must incorporate the functions provided by the corresponding layer for each network. Although in theory the functionality for each layer should be more or less the same regardless of the network, in practice it is not the case. Even high level protocols make

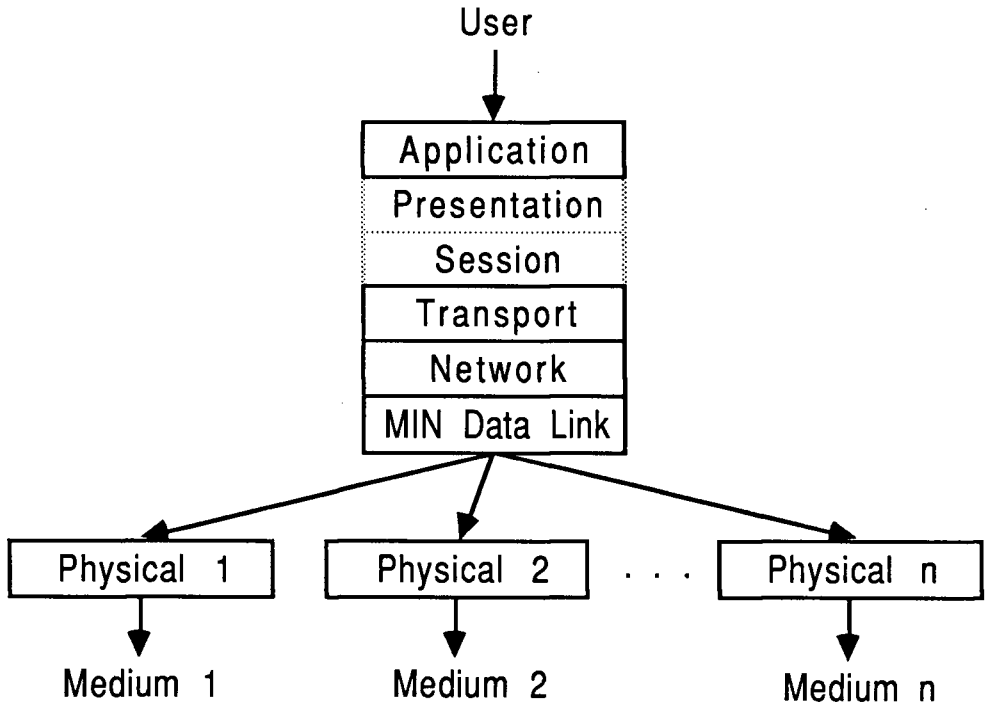


Figure 2.1: Routing Decision with MIN in Data Link Layer

assumptions about the underlying network. For instance, BSP is a transport level protocol designed specifically for the Cambridge Ring. Additionally, each protocol also makes assumptions about the supporting protocol, or in other words, the layer directly beneath it. For example, TCP assumes IP to be the Network protocol, while BSP assumes BB to be underneath it. Furthermore, the MIN Data Link layer would need to know how to handle all possible packet formats (different for each network). Each future addition of a new type of network would require a modification to this layer as well.

The same arguments may be applied to the Network layer, the Transport layer, and each successively higher layer. However, they become less significant in the higher layers because they are application dependent rather than network dependent.

Another consideration is that we wish to let the user optionally select which networks to use for sending a message. As stated earlier in Section 1.2, each application should know which network it is most suited to, and should therefore be allowed to make that decision if so desired. If MIN is placed in the Data Link layer, then the user's path specification would need to be passed through all the higher layers, since the user process may only access the Application layer. Furthermore, allowing the user to choose the networks implies that each "MIN connection" between two user processes may have different characteristics in terms of which networks are used. Thus MIN should provide process-to-process connections, rather than a single host-to-host con-

nection which multiplexes process-to-process connections provided by a higher layer. This suggests that MIN should be located at least above the Transport layer. Also, by being above the Transport layer MIN is provided with reliable connection-oriented communication, thereby making its task easier. The supporting protocols will perform error detection and recovery as appropriate to their respective layer.

Functionally, MIN corresponds to the Session layer, in that it enables a user process to establish a connection, or session, with another process. However, in our environment a "MIN connection" may actually consist of multiple connections or sessions, one on each of the networks which the user process wishes to use. A Session "connection" in the ISO sense appears to imply the use of only a single network; there is no provision for the case of using multiple networks simultaneously (one may point out that if gateways are involved then the "connection" would include more than one network; however in that case the networks are used serially, not simultaneously; furthermore the internetworking is handled in the Network layer, and is transparent to the Session layer).

As for the Presentation layer, there does not seem to be any motivations, functional or otherwise for placing MIN there. This leaves the Application layer. From the practicality viewpoint, this would be the best place for MIN since the least number of layers would be affected. It is also appropriate in that MIN is intended as an interface for user processes. The path selection parameter would then be obtained directly from

the user, without having to pass down through various other layers. However, it is not really part of the Application layer's function to be concerned with connections.

We must point out however, that the decision of whether to place MIN in the Application layer or the Session layer is not a critical one. It depends on the particular implementation environment. If a new system is being created which will contain all seven layers of the ISO model, then MIN should probably be placed in the Session layer, since this is functionally more correct. However, if MIN is being incorporated into an existing system, then putting it into the Application layer would make the job easier. Moreover we note once again that many existing systems do not contain the Presentation and Session layers, so that whether MIN is conceptually placed in the Application or the Session layer makes no difference in the implementation. For the remainder of this chapter we will assume that MIN is placed in the Application layer, as illustrated in Figure 2.2.

2.2 Service Model

In the previous section we have implicitly assumed that MIN provides a connection-oriented interface to the user process. The reasons for this choice were purposedly left out from the earlier discussion because they did not really relate to the topic of where MIN should reside in the ISO model. Mentioning them would only have served to confuse the reader. The service model for MIN is discussed in this section.

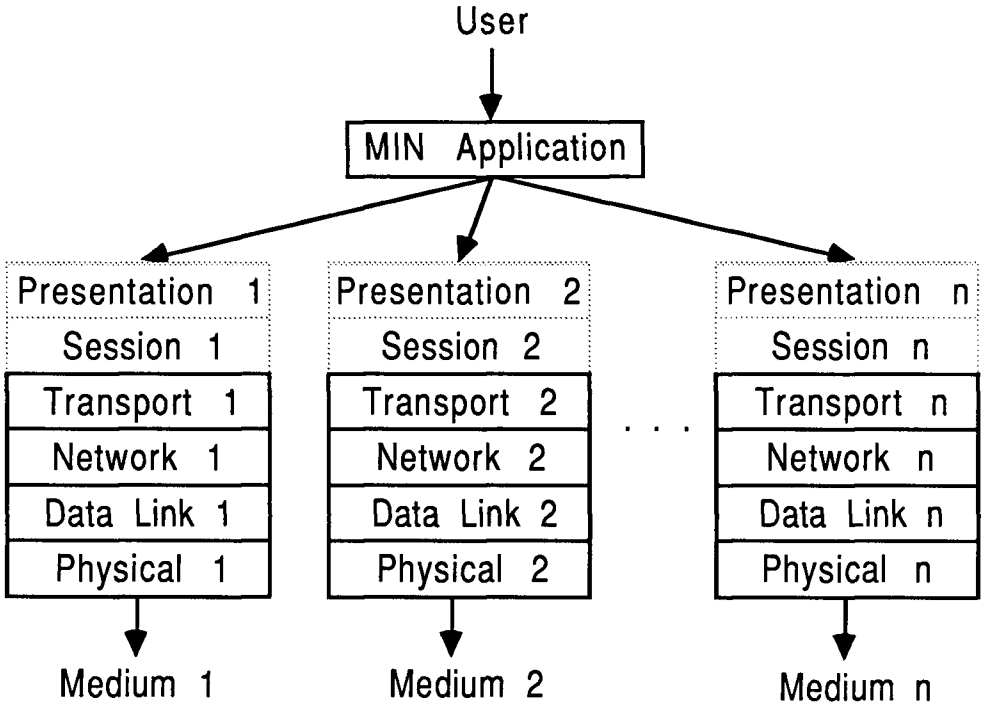


Figure 2.2: Overall Structure with MIN in Application Layer

There are two basic types of remote inter-process communication (IPC) : connectionless (datagram) and connection-oriented (virtual circuit) [Tane81]. In the connection-oriented type of remote communication, a logical association is established between two processes which wish to exchange data. This virtual connection provides a reliable service in that lost messages are detected, and messages are guaranteed to be delivered in the same order in which they were sent. When one of the two processes wants to send a message, it is only necessary to supply a reference to the logical connection, rather than the receiving process's complete identification (host id + process id). An example of a system which provides virtual circuits as one of its communication services is 4.2 BSD Unix, which calls each end of the connection a "stream socket" [Leff83].

In a connectionless IPC environment, each time that a message is to be sent, the destination process' complete identification must be supplied. 4.2 BSD Unix also provides this type of service, in the form of an Unreliable Datagram Protocol (UDP). Here the user utilizes a "datagram socket", rather than a "stream socket" as in the connection-oriented case.

A connection-oriented IPC model offers a higher level abstraction of communication to users. It provides a reliable end-to-end service, whereas in the connectionless model there is no concept of message sequencing, so that the user would need to deal with lost and out-of-order messages himself. More work is required on the system's part, though, to provide connection-oriented IPC, as the state of each connection must be

maintained. With a connectionless model the transmission of each message is a totally independent operation from the system's point of view. Nonetheless most systems try to provide a connection-oriented type of service, since it offers a simpler interface to the user. Therefore a connection-oriented communication model was chosen for MIN,

2.3 Message Versus Byte-Stream IPC

For a connection-oriented service there are two possible formats for message exchange. One method views each transmitted message as an individual entity, so that one message is delivered to the receiver at a time, in the same size as it was sent. The other method views the connection as a boundaryless byte stream, so that the receiver may accept the data in quantities different from the sizes of the messages sent by the sender. This style of data transfer corresponds to the file I/O model used in Unix, and is, in fact, the interface adopted for the stream sockets in the 4.2 BSD system.

The differences in these two methods affect how messages must be handled at the receiving end, and not so much the transmission of messages. Whichever of these two methods is chosen for a system, though, the user may choose to impose his own view on incoming data, by building his own interface on top of the system services. For instance, if the system delivers a message at a time, but the user wants a byte stream abstraction, then he can simply keep receiving individual messages until the desired number of bytes have arrived. Any leftover data in a partially consumed message can

be stored for a subsequent "stream receive" operation.

Stream Receive :
If any data saved from previous message
Then copy as much as is requested into user buffer
While request not satisfied
Await arrival of a message
Copy as much as is requested from message to user buffer
If message is only partially consumed
Then save remaining data for next call

Figure 2.3: Stream Receive for Message-oriented Service

The other possiblity is that the system provides a boundaryless byte stream but the user wants to preserve message boundaries. In this case, the user must insert his own message delimiters. This can be achieved either through the use of a special pattern as a message terminator, or by pre-pending each message with a header that specifies how large the following message is.

The use of either message terminators or message headers adds overhead data, and therefore causes additional processing overhead and message transmission delay. Also, the message-oriented style of message exchange is applicable to more types of communication than the byte stream method, which is suitable mainly for file or bulk transfers. The message-oriented method is therefore the style chosen for MIN.

```
Message Receive (using terminators) :
        Intialize message size counter to zero
        Repeat
                Get next byte
                If it is a message terminator
                        Then stop and return message size
                Copy byte into next location in user buffer
                Increment message size counter
        Return message size


Message Receive (using headers) :
        Get bytes comprising header
        Get as many bytes as message size specified in header
        Copy bytes into user buffer
        Return message size
```

Figure 2.4: Message Receive for Byte Stream Service

2.4 MIN Primitives

Next the set of primitives offered to the user must be designed. For a connection-based service, the usual model of communication between processes is client-server. This is the model we will use for our primitives. Also, we strive for parsimony, since too many primitives would confuse users. However, we must ensure that these primitives provide sufficient functionality. The basic requirements are to enable users to establish and disconnect MIN connections, through which they may send and receive messages of

arbitrary size.

For a client-server model, connection establishment is initiated by the client, who is the active party. The server waits passively for a connection request. Thus two primitives are required for connection establishment :

MINid = **connect(servername, networks)**

MINid = **accept(servername, networks)**

Connect is used by the client to try to establish a MIN connection with the server whose name is **servername**. Symbolic identifiers are used because MIN is an interface for user processes. Since **servername** is a symbolic identifier, we are assuming the existence of a name service which maps server names to addresses. This server address may be a Session layer address, in which case MIN can simply pass it along to the Session layer when it tries to establish a session on each network (see Section 2.6). However, if the Session layer for each network uses a different address, then MIN must map the server address to a set of Session layer addresses. In other words, the name server takes the symbolic identifier **servername** and returns an Application layer address to MIN, which then maps it to a set of Session layer addresses, one per network. Alternatively the name server may directly map **servername** to the set of Session layer addresses. Each Session layer address should also include the id of

the host on which the server process resides, since we assume server locations to be transparent to clients. If this were not the case for a particular system, then **connect** must take an additional parameter for specifying the server's host.

Accept is used by the server which wishes to accept a MIN connection request from a client. When this request is issued, the name **servername** is logically bound to the server process. The binding remains for the duration of the process. Only one process may be associated with a symbolic name at any given time. Again a name service is assumed which enforces this property. However, this uniqueness property could be enforced per-host or per-system. For implementation purposes it would depend on the name service available with the host system. If none were available, then we would recommend the implementation of one which enforced system-wide server-name uniqueness, since it would allow remotely accessed servers to have freedom of location.

Accept only accepts one MIN client's connection request. The server must call **accept** each time that it wishes to accept another MIN connection. The **accept** primitive blocks the invoking server process until a MIN connection request is successfully accepted or some error occurs during the connection establishment phase (see Section 2.6 for more details).

For both **accept** and **connect**, the second parameter **networks** specifies the network or set of networks which the **accept** or **connect** operation is to include. This raises the issue of network identification, which will be discussed in Section 2.5. For

the time being, simply assume that an identification scheme exists which can be used for **networks**. When the user does not care which networks are to be used for the MIN connection, the parameter **networks** can simply be omitted. In that case the decision of which networks to use is left to the MIN software. MIN will attempt to use all the networks available. Most users would, in fact, omit the **networks** parameter, especially for **accept**. However, for generality the **networks** parameter is included for **accept** as well as for **connect**. This implies, though, that the **networks** specified by the client may not be the same as the **networks** specified by the server. This is resolved by a simple handshaking procedure which is discussed in Section 2.6.

Both **accept** and **connect** return a non-negative MIN connection id **MINid** when the operation is successful. Otherwise a negative value indicating the cause of error is returned. Some of the possible errors are :

1. **networks** is invalid
2. unknown **servername** (for **connect** only)
3. **servername** already in use (for **accept** only)

Other errors may occur in dealing with the system's Session level services. These are system-dependent.

The MIN connection id returned by **connect** and **accept** is a logical identification of the connection just established. This **MINid** is used for all subsequent operations on that connection, such as sending and receiving messages (described below). Since

the assignment of a MIN connection id is done at both ends of a connection, this implies that two ids are created for each connection - one returned to the server from an **accept** call, and another returned to the client from a **connect** call. Therefore some means is necessary for associating the two ids, so that messages sent by the server, using say id1, will be received by the client using say id2, and vice versa.

There are two possible ways of dealing with this problem. One method is to keep track of all the MIN connection ids being used in the entire distributed MIN environment. Then, each connection can be assigned a unique id which is used to reference it at both ends. In other words the server would get back from the **accept** call the same **MINid** as that returned to the client on the **connect** call. However, this solution is difficult to implement, as the dynamic creation and destruction of connection ids must be made known to all of the hosts.

The alternative scheme requires a per-connection association of MIN ids to be maintained locally. That is, suppose a connection between a process on machine1 and a process on machine2 is assigned id1 on machine1 and id2 on machine2. At machine1 the local system can record that the id at the other end of this connection is id2, while at machine2 the local system can record that the other end of this connection has id1. This method is preferable since it only involves the two hosts at the two ends of a connection, rather than all hosts in the entire MIN environment. There is still the problem of how each side of the connection discovers what the id being used at

the other end is. This may also be simply resolved during the handshaking done at connection establishment time mentioned above (see Section 2.6).

To exchange messages, two primitives are needed :

result = **send(MINid, message, size, networks)**

result = **receive(MINid, buffer, size)**

The first parameter needed for both **send** and **receive** is the MINid returned from a successful **connect** or **accept** call. For **send**, **size** is the size of **message** in bytes. Since the entire message is considered as a single unit, **send** only returns whether the transmission attempt was successful. If this MIN connection involves more than one network connection, then one of them is chosen for sending the whole message. The user may limit the networks to choose from through the **networks** parameter. The format of **networks** is the same as that for **connect** and **accept**. However, if the **networks** supplied to the **send** call includes a network which was not specified in the **connect** or **accept** call which established the connection, an error is returned and the message is not sent. The **networks** parameter may be omitted, in which case the routing decision is left entirely to the MIN software (see 2.7). Although the user message is treated as a single entity and sent over just one network, it may require more than one actual transmission. This depends on the maximum message size allowed by

the chosen network's supporting software.

The **send** primitive is non-blocking, so that the user does not have to wait for the message to be received by the other process, nor for an associated reply message to arrive before continuing execution. However, in an actual implementation there is usually some delay from the wait for acknowledgements by the supporting layers of protocols. A successful **send** does not imply that the message was actually received, merely that it was delivered and available to the destination process. A non-blocking style of **send** was chosen in order to try to attain higher performance. With more than one supporting network for the MIN connection, it is possible to have more than one message in-flight simultaneously, if the **send** primitive is non-blocking.

For **receive**, **buffer** is the location of a buffer **size** bytes in size in which to place the next incoming message. If the arriving message is too large for the buffer, then the message is not placed in **buffer**. If the buffer is big enough, then it is filled with the message's contents. In any case, though, the size of the message is returned as the **receive** primitive's return value. If the buffer was big enough, then this return value will let the user know how much data he has received. If the buffer was not big enough, then this return value will let the user know how large of a buffer is needed for his next **receive** attempt. A more detailed discussion on the message reception procedure appears in Section 2.9.

One final primitive is needed for removing MIN connections :

result = **close(MINid)**

This primitive may be used by clients or servers. Only one parameter, the MIN connection id, is needed. This causes all underlying network connections used for this MIN connection to be released. In addition any data which has arrived for this connection but has not yet been passed to the user is discarded. However, a **close** call does not affect the other end of the connection, in that the process at the other end may still **receive** any unreceived data which was sent from this end before the **close** took place. Note that, as stated earlier, if **MINid** was established from an **accept** request, closing the connection does not unbind the process from **servername**. This does not occur until the termination of the server process.

2.5 Host Reachability

In order for two processes to connect to each other for communication, some physical path must exist between the hosts which the two processes reside on. With the use of bridges and gateways, this is possible even if the two hosts are not both connected to the same network. This can lead to a very wide-spread inter-connection of hosts,

and combine to form a Long Haul Network like situation, where a message must pass through a number of intermediate hosts before reaching its eventual destination.

In this thesis, though, we restrict our environment to exclude the use of bridges and gateways. That is, two hosts are considered reachable from each other only if they share at least one LAN in common. For instance, in Figure 2.5, host B is considered to be reachable from host D, but host D is not considered to be reachable from host A, as far as our MIN environment is concerned.

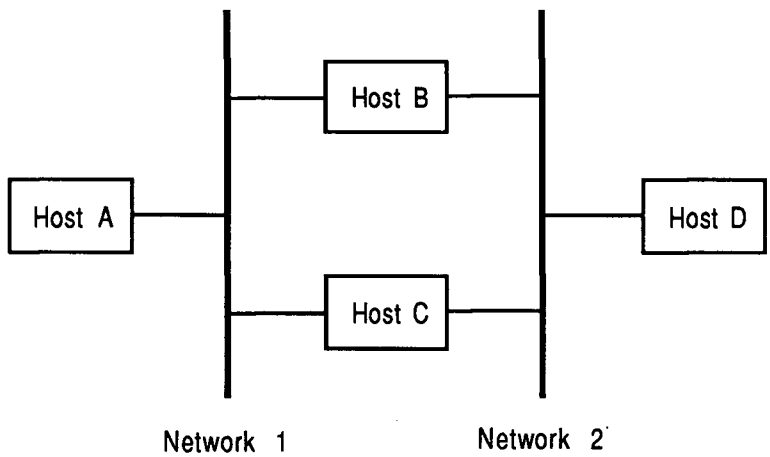


Figure 2.5: MIN Multiple Network Domain

This restriction was imposed mainly to simplify our problem, not so much due to how bridges and gateways affect host reachability, but because of how they affect path selection based on network loads (discussed in Section 2.8). For host reachability, we can simply view those networks which are interconnected by gateways as being one

large network. In Figure 2.5 for instance, suppose Host B has a Network layer which bridges Network1 and Network 2. We can thus consider Networks 1 and 2 to be a single network which connects Hosts A and D. That is, from Host A's point of view, it can reach Host D via Network 1, while from Host D's point of view it can reach Host A via Network 2. This can be recorded in the host reachability data structures (to be discussed shortly) in Host A and Host D. However, the fact that a bridge is involved should be transparent to MIN, since it is handled in the underlying Network layer. In any case, though, before the MIN concept can effectively include the use of gateways, the difficult problems which would be caused for path selection must be satisfactorily dealt with.

Not all of the hosts need to be connected to all of the networks in our environment. For example, in Figure 2.5 Host A and Host B can only communicate through Network 1, but Host B and Host C can reach each other through both Network 1 and Network 2. Thus a means for determining which hosts can reach each other, and along which networks, must be devised. That is, a function which takes a (host, host) pair as input and returns a set of networks on which the two hosts can reach each other is required.

Since the domain for the (host, host) pair is the set of valid host ids, the issue of host identification is relevant to our problem. Unfortunately, it is a complex issue which can not be adequately examined here. A thorough treatment can be found in [Chan86], which discusses the properties desirable for host identification schemes, and

presents one which satisfies these properties. For our purposes, we will simply assume the existence of a satisfactory scheme.

The output of our mapping function has the set of valid network identifiers as its domain. This raises the issue of network identification. In this case, though, since it is only relevant to our own environment of multiply-connected machines, all we need is an internal scheme for uniquely identifying each network. A simple enumeration method, which assigns a static id to each network, suffices. For communicating with outside networks through bridges or gateways, we simply view our entire MIN environment as a single LAN, and just give it one global network identifier.

One approach to implementing the host reachability function is to maintain a data structure which contains all the mapping information. An obvious candidate for this data structure is a two dimensional matrix. The contents of the matrix could be organized in two different ways. One method is to take the point of view of the host on which the structure is maintained. That is, it will store for each network the set of hosts which are reachable **from this host** along that network. This implies that a different structure would be needed for each host. The second method is to take into account our entire MIN environment, and store for each network all the hosts which are connected to that network. As shown in Figure 2.6, with this scheme entry A_{ij} is marked **yes** only if host j is connected to network i. The first method is more efficient, since it only requires looking up the entries for the destination host. However,

the second method is much more flexible since the same structure can be used for all hosts.

Host / LAN	1	2	3	4
1	yes	yes	yes	yes
2	no	yes	yes	yes
3	yes	yes	yes	yes
4	no	yes	no	yes

Figure 2.6: Host Reachability Matrix

The information in the structure can be maintained statically or dynamically. If it is done statically, then only the physical topology is taken into account, so that modification of the structure's contents are only necessary when hosts or networks are added or removed. With dynamic maintenance, the goal is to maintain the most up-to-date status information, so that it is necessary to keep track of the crashes and recoveries of both hosts and networks. With a static scheme, a user will only discover the loss of a host or network when his attempt to send a message fails. This usually occurs after some system-dependent number of timeouts and retries. Thus

one advantage of the dynamic scheme is that it would save the user from time wasted in finding out the loss of a host or network. However, the detection of host loss or recovery is difficult, and usually requires the sending of probe messages by the system. Since host and network crashes are not very common, a static scheme is sufficient for our purposes.

Even for a static scheme, though, a simple matrix is actually not a totally satisfactory data structure. This is due to its fixed dimensions, which do not allow easy growth or shrinkage. Since most hosts can join a network unobtrusively, it would be advantageous if the data structure can be readily enlarged. One structure which is suitable is shown in Figure 2.7. It consists of a linked list of host ids, each of which holds a pointer to a linked list of network ids. This structure has much greater freedom to grow and shrink in all directions. It also allows the set of networks common to two hosts to be easily found : this is simply the set of networks which appear in the linked list of networks for both hosts.

2.6 Connection Establishment

The set of networks to use may be specified by the user for both the **connect** and **accept** primitives. This means that the client and server may select different sets of networks. For example, the server may be accepting on networks 1 and 2, while the client is trying to connect on networks 2 and 3. In this example, connection can only

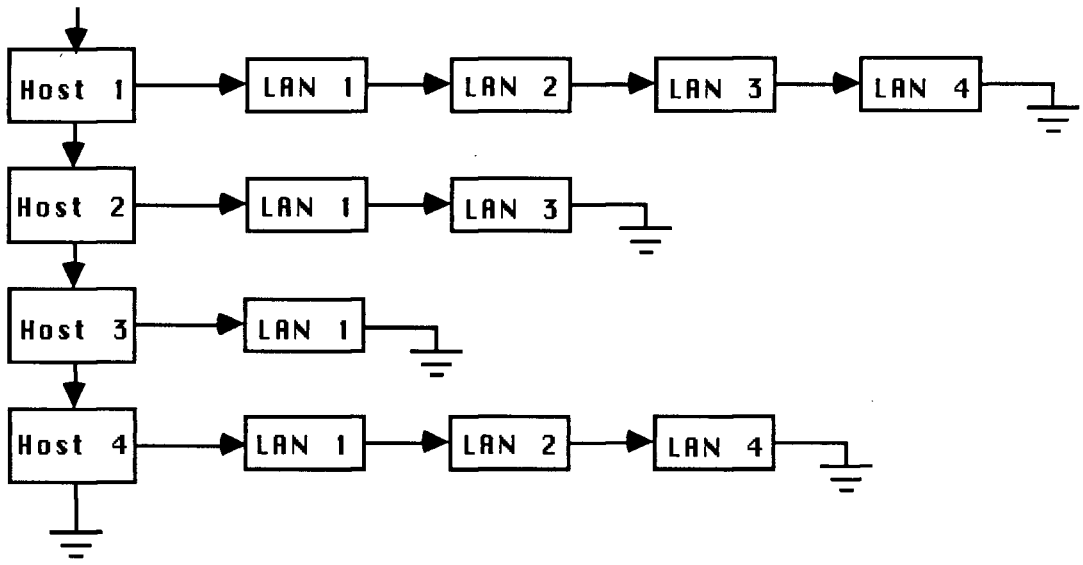


Figure 2.7: Host Reachability List Structure

be established on network 2, since it is the only network common to both the client's set and the server's set of networks. With n networks, the number of possible values for the **networks** parameter is 2^n. The total possible number of (accept-networks, connect-networks) combinations is 2^{n^2}. The MIN primitives must therefore perform some handshaking during connection establishment, so that both sides will have the same view of the MIN connection before any messages are exchanged.

If the client attempts to connect on a network on which the server is not accepting requests, then the connect attempt will fail. Therefore the client, after making a connection attempt on each of the networks specified in the **networks** passed in the **connect** call, will know exactly which networks connection was established on. He

can then send an "end-of-connect" notification to the server on one of the connected networks (any one will do), to inform the server that he has finished his MIN connection attempt. The server, upon receiving this notification, can stop awaiting reception requests on the remaining networks, if any. The MIN connection phase for the client and server are shown in Figures 2.8 and 2.9 respectively.

```
MIN Connect :
    Verify that server is reachable on each network specified
        in networks, removing any which are not
    If no valid network in networks
        Then return error
    Assign a MINid for this MIN connection
    For each valid network specified in networks do
        Try to establish a connection on that network
        If successful
            Then include this network for this MIN connection
    If connection was not established on any network
        Then free this MINid and return error
        Else (connection established on at least 1 network)
            Send end-of-connect notification containing MINid to server
            Wait for end-of-accept reply with MINid used by server
            Return MINid to invoking client
```

Figure 2.8: Connection Phase for Client

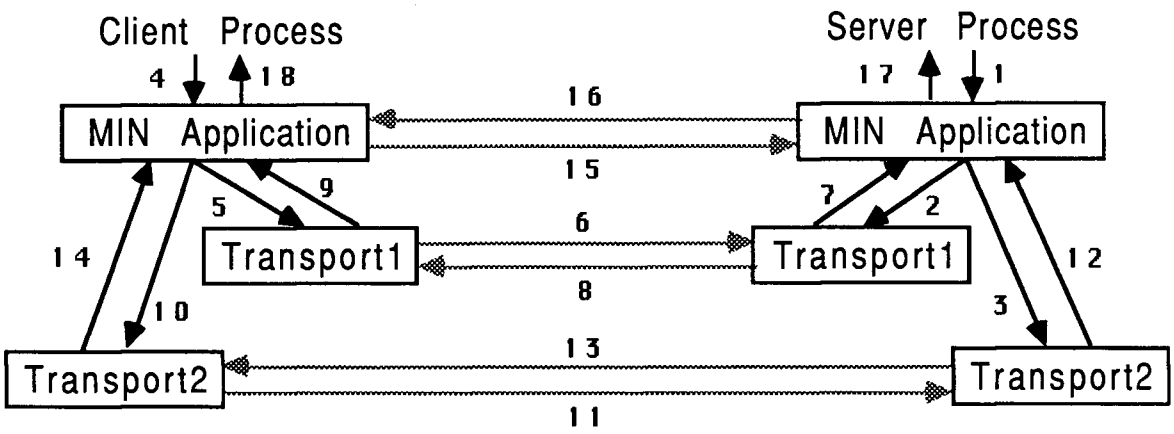
As mentioned in Section 2.4, each side of the connection must inform the other side of the connection of the MIN connection id which it is using. This is conveniently done during the handshaking procedure just described. In the end-of-connect notification

```
MIN Accept :
    Verify that server is reachable on each network specified
        in networks, removing any which are not
    If no valid network in networks
        Then return error
    Repeat
        Await arrival of a connection request or end-of-connect
            notification on any valid network specified in networks
        If it is a connection request which arrived
            Then accept the request and include
                this network for this MIN connection
            Else (it is an end-of-connect notification)
                Record MINid sent by client in end-of-connect notification
                Send end-of-accept reply with server's MINid
                Return MINid to invoking server
```

Figure 2.9: Connection Phase for Server

sent by the client, he can include the MIN connection id which he is using. The server, upon receving this notification, sends an end-of-accept reply containing his own MIN connection id. This is also shown in Figures 2.8 and 2.9. To clarify further, Figure 2.10 gives an event-by-event example of the connection phase. Since the Presentation and Session layers are usually not present, we have assumed them to be null and made MIN deal directly with the Transport entities for each network.

Connection ids must be uniquely assigned within each host. That is, there must be no more than one MIN connection which is logically identified by a particular id. To



1. Server process invokes MIN primitive **accept** for Networks 1 and 2
2. MIN requests a passive Open from Transport layer for Network 1 (Transport 1)
3. MIN requests a passive Open from Transport layer for Network 2 (Transport 2)
4. Client process invokes MIN primitive **connect** for Networks 1 and 2
5. MIN requests an active Open from Transport layer for Network 1 (Transport 1)
6. Transport 1 (Client host) sends Open request to Transport 1 (Server host)
7. Transport 1 (Server host) accepts Open request and returns Success to MIN
8. Transport 1 (Server host) replies to Transport 1 (Client host) with Open Success
9. Transport 1 (Client host) returns Success to MIN
10. MIN requests an active Open from Transport layer for Network 2 (Transport 2)
11. Transport 2 (Client host) sends Open request to Transport 2 (Server host)
12. Transport 2 (Server host) accepts Open request and returns Success to MIN
13. Transport 2 (Server host) replies to Transport 2 (Client host) with Open Success
14. Transport 2 (Client host) returns Success to MIN
15. MIN (Client host) sends End-of-connect notification with client MINid to MIN (Server host)
16. MIN (Server host) sends End-of-accept reply with server MINid to MIN (Client host)
17. MIN (Server host) returns server MINid to Server process
18. MIN (Client host) returns client MINid to client process

Figure 2.10: Event-by-Event Example of MIN Connection Establishment

ensure this, a 32 bit integer counter is used for assigning MIN ids. When a **connect** or **accept** call is successful, the current value of the counter is returned as the MIN connection id. The counter is then incremented, so that the same value is not used again. Although the counter will eventually wrap around, 32 bits should provide a large enough domain to guarantee uniqueness. One counter is used per host, for all MIN connections in that host. Additionally, some state information must be maintained for each MIN connection, for use in message exchange. This is discussed in more detail in Section 2.9.

It is assumed that the underlying communication services for each network provides reliable connection establishment. It is also assumed that it is possible to simultaneously wait for the arrival of a connection request on one network, and for a message (end-of-connect notification) coming in on another network on which connection was already established.

2.7 Path Selection

In sending messages on a MIN connection which involves multiple underlying networks, path selection is performed on a per-message basis. That is, different messages may travel different paths, but no attempt is made to subdivide a user message. The choice of which network to send a message is made to optimize performance. Two important

criteria to consider in making this decision are the current load on each network and the message size [Vuon83]. More criteria may be found to be of significance in the future, therefore the decision algorithm or formula should be easily extendible.

One possibility is to prioritize the criteria. That is, the one which is deemed as most important is always used first. If it is not sufficient to select a network (there is a tie between two or more networks), then the second most important criterion is used to decide between the contenders, and so on. This method is extendible, as any new criterion can be inserted into the appropriate position in the order of evaluation, according to its relative importance. However, it may not be a very good algorithm since it usually ignores all but the most important criterion.

Another method is to quantify the suitability of each network for each criterion, then calculate the overall suitability value of each network using a weighted sum.

For each network :

$$v = w_1 p_1 + w_2 p_2 + ... + w_n p_n$$

where

p_i = suitability value with respect to criterion i

w_i = relative weight of suitability value for criterion i

v = overall suitability value of this network

$$\sum_i p_i = 1$$

For example, suppose message size is the criterion to use. If we know that Ethernet is good for message sizes > 500 bytes, but is bad for message sizes <= 20 bytes, and that the Cambridge Ring is good for message sizes <= 100 bytes, but is bad for message sizes >1000 bytes, then we can set up a table like that in Figure 2.11.

Size of Message \ LAN Type	Ethernet	Cambridge Ring
<= 20	0.0	1.0
20 < <= 100	0.3	0.7
100 < <= 500	0.5	0.5
500 < <= 1000	0.8	0.2
1000 <	1.0	0.0

Figure 2.11: Suitability Quantification Using Message Size

This method too is extendible, as new terms can be easily added to the formula to account for new decision criteria. The relative weights, as well as the quantification scheme for each criterion, can be tuned according to measurement experiments for each particular system. However, this may be difficult when more than two criteria are used. Also, it is not as efficient as the priority method, since all criteria are always

considered, and each involves a multiplication. Efficiency is a concern since the path selection must be made on the sending of every user message. If the maximum benefit from choosing the "best" network, in terms of decreased delay time, is **t**, then the decision algorithm must require less than time **t**, or the potential benefit is lost.

2.8 Load Monitoring

An important criterion for selecting a path for sending a user message is the current load of each network. Since the transmission medium in most LANs is shared, having many stations which all wish to send messages around the same time would imply a longer delay before each station can access the medium. Delay can also come from the queuing of outgoing messages with each station, at various protocol layers. Thus the overall delay can be attributable to the global load and the local load.

Message transmission in most LANs is broadcast in nature (true for bus and ring topologies, but not for star). Thus the global load can be determined at each station by listening for any activity on the medium. For example, in a slotted ring, a station can monitor the number of filled slots passing by, regardless of the contents' destination. However, this is difficult without hardware support. Many vendors offer interfaces which only provide transmission and reception capabilities, but not monitoring functons.

The local load is also difficult to measure, since queueing of messages, packets, and frames is dispersed throughout the different protocol layers. However, all must eventually pass through the physical layer, be they sends or receives. Therefore it is possible to get an estimate of the current local load, as well as make a prediction of what it will be in the near future. To do this, time can be divided into a number of fixed length quanta. The number of packets sent and received in each quantum can be counted. Using the **n** most recent quantum counts, a prediction can be made of the next quantum count. This predicted value could then be used as an approximation of the current load. Greater weight can be given to the more recent counts since they represent more up-to-date information.

For each network :

$$l = w_1 x_1 + w_2 x_2 + ... + w_n x_n$$

where

x_i = packets sent and received in the i-th previous quantum

w_i = relative weight for the i-th previous count

l = predicted quantum count for the next quantum

$$\sum_i x_i = 1$$

To check the accuracy of the prediction, each predicted load could be compared against the actual count measured in the next quantum. The relative weights can then be adjusted to reflect the characteristics of the system. Dynamic weight adjustment schemes, however, are extremely expensive.

The quantum length must be carefully chosen. If it is too large, then accuracy is lost. On the other hand, if it is too small, then heavy processing overhead will be incurred since a load prediction is required at the end of every quantum. A similar tradeoff exists for choosing the number of quantum counts to use in predicting the next one. Too few would be insufficient to reflect a trend, and too many would incur memory as well as processing overhead. Again tuning is in order.

Unfortunately this method only provides some idea of the local load. Additionally, it may even give the wrong information during times of very heavy network loads. This would occur if the network is so heavily used that a rising load would allow each station to perform less and less sends and receives during each time quantum. Our scheme would report a decreasing local load, and therefore infer an incorrect assessment of the network situation. Furthermore the work involved in calculating a weighted sum for every time quantum is very heavy. Therefore a simpler strategy based on the hot potato algorithm may be a better plan. Here the load for a network is simply

interpreted as the length of the sending queue at the physical layer. This algorithm requires much less computing time, and produces reasonably accurate results. Since the aim in determining the network load is to send a message quickly by using the lightest network, and the length of the outgoing queue for a network contributes to the delay before a message can be sent on that network, then it makes sense to consider the length of the outgoing queue.

The length of the outgoing queue may be measured in terms of the number of packets waiting to be transmitted, or the total number of bytes in all of these packets. It is unclear which of these would be better to use. Also, in either case the length value may need to be multiplied by a factor which is different for each network. The values of the factors depend on the relative speeds of the networks. That is, if for example the Ethernet is twice as fast as the Cambridge Ring, then one packet in the Cambridge Ring queue should be considered as being equal to two packets in the Ethernet queue. Measurements could be taken in order to obtain some idea of the relative speeds of the networks to be used. However, since performance is affected by load, the factor would need to be dynamically adjusted, which could be costly. The adjustment for the factor may need to be different for each network as well, since different networks respond differently to changes in load. It is beyond the scope of this thesis to look thoroughly into the problem of quantifying the relative speeds of different networks. We merely point out that this issue needs to be considered.

The length of the incoming queue could, under certain circumstances, provide some information on the global network load. This would be true if the messages sent on the network are fairly evenly distributed among all the hosts as destinations. However, since this not always the case, the length of the incoming queue is not used by the MIN at all. There are other possible means for obtaining some indication of the global network load using the physical layer, even without adequate hardware support. This would depend, though, on the underlying network technology. For example, in a token bus or ring the length of time between successive arrivals of the token can be measured. The longer the time, the greater the probability that the token was consumed during that time. For a slotted ring, we can similarly measure the length of time between successive arrivals of an empty slot. For a CSMA/CD bus, it may be possible to count the number of collisions (and therefore re-transmissions), and include this value in the quantum count. In all of these cases, additional processing overhead is incurred. The optimum would still be a hardware interface which automatically monitors the network load, and makes this information available to the software as a value between 0 and 1.

The load information is collected in the physical layer, but it is made use of many layers above, in the MIN software. Still remaining then is the question of how to pass the information upwards. If it is accessed directly by MIN, then it would violate the protocol design principle which stipulates each layer to be aware of only the layer above and the layer below. However, the alternative of filtering the information through each

of the intervening layers is highly unattractive. It would be slow, as well as require each layer to be modified to accomodate this passing of load information. This change would need to be made to every protocol layer for every network! Accessing it directly is therefore a much better idea. To lessen the violation of the layered protocol principle, a **Network Monitor** could be utilized for gathering the load information from all the networks. Then MIN would simply obtain the information from the Network Monitor, which it can perceive as just another system entity. The Network Monitor may be either a separate process, if the host system adequately supports local inter-process communication, or it may be a shared module of system routines. The addition of the Network Monitor completes the picture of our overall software structure, as shown in Figure 2.12.

Finally, we must discuss how the inclusion of gateways affects the task of path selection based on network loads. As alluded to in Section 2.6, this is not a simple problem. Suppose a host A can reach another host B either via Network 1 or via Networks 2 and 3 which are bridged by a host C. If network loads are to be used in deciding between the two possible paths, then we must consider the load on Network 3 as well as the load on Network 2 for the second path. Since the delay for a message to travel through Network 2 is independent to that for Network 3. intuitively we should add the load for Network 3 to the load for Network 2. Additionally, we must consider the processing delay for the Network layer, the Data Link layer, and the Physical layer

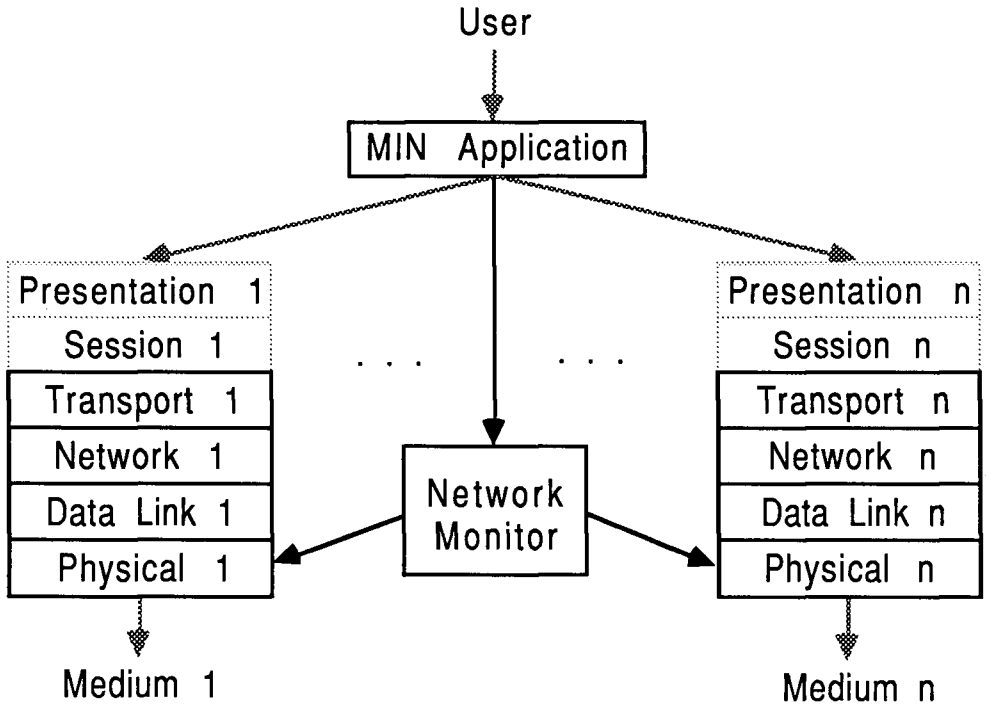


Figure 2.12: Network Load Monitor

in the gateway host C. This is difficult to determine. Furthermore, how should the load information for Network 3 be made available to the MIN software on host A, where the path selection decision is being made? Since host A is not connected to Network 3 this load information must be somehow passed along, probably through host C. If a path includes more than 1 gateway then the problem of accessing remote load information for the source host becomes even more complex. The inclusion of gateways in a multiple network environment is a topic which is beyond the scope of this thesis.

2.9 Message Exchange

To send a user message on a MIN connection which consists of more than one physical network connection, one of the networks is selected for the actual transmission, as discussed earlier. However, there is no guarantee that the transmission attempt on the chosen path will actually succeed. A failure may arise from the loss of that network, or from the crash of the destination process's host, or from the death of the destination process itself. Since the cause is not easily determined, we will be pessimistic and assume that it is due to a network failure, and re-transmit the user message on another network. We can only conclude that it is a host failure, or equivalently, the death of the receiving process, after a transmission attempt on every network for this MIN connection has failed. The sending algorithm is shown below.

```
MIN Send :
    Construct a MIN message header for this message
    Repeat
        Choose network most favourable for sending on
        Try to send the MIN header on that network
        If failed
            Then disassociate this network from this MIN connection
        Else
            Try to send the user message on that network
            If failed
                Then disassociate this network from this MIN connection
    Until header and message successfully sent
        or have attempted and failed on all networks
```

Figure 2.13: Message Transmission

Notice that a transmission error on any network will cause that network to be disassociated from the MIN connection. This means that that network will never be used again for message transmission or reception, for the remaining duration of this MIN connection. The assumption is that network failures usually last more than a brief moment, so that it would be a waste of time to make any further transmission attempts on this network for sending subsequent user messages. Nonetheless, it would be beneficial if this disassociation could be temporary instead, so that the failed network can be used again once it has recovered. One way of doing so is through the Network Monitor. When a network is detected as being out-of-service, the Network Monitor can set the suitability value for that network as negative infinity, so that it will never be

selected for transmission. Once that network has recovered, its value can be returned to normal. However, this scheme requires additional system support, since a network failure must be accurately detected and distinguished from other possible causes of communication failures, such as the crash of the destination host.

In Figure 2.13 a MIN message header is mentioned, which has not been discussed as yet. Its purpose is relevant to the task of message reception, which is described below.

To receive a message on a MIN connection which involves multiple networks is slightly more difficult. Since multiple paths are available, and the send primitive is non-blocking, it is possible for user messages to arrive out of sequence. This property is purely a consequence of the behaviour of MIN's non-blocking transmission of messages. Therefore we must deal with message sequencing ourselves. For each MIN connection, it is necessary to maintain a sending sequence number and a receiving sequence number. The sending sequence number is put into a MIN header which is prepended to each user message to be sent. Also required is the size of the message, and the MIN connection id used by the receiving side of this connection. This simple header format is shown in Figure 2.14. Additional contents may be placed in the header in the future, as new features are incorporated into MIN.

In order to avoid excessive data copying, the MIN header is not physically prepended to the user message and sent off together. Instead, the header is sent first separately,

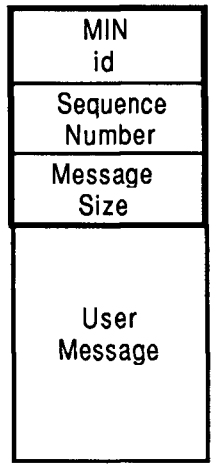


Figure 2.14: MIN Message Header

and then the user message is sent immediately afterwards. However, both the header and its associated user message must be sent on the same network. For each network involved in a MIN connection, the software must keep track of whether a MIN header or its following user message is due to arrive next. As long as each header/message pair is sent on the same network, one after the other, then this is not a difficult task, since the arrival order on each network will always be : header, message, header, message, and so on.

On the arrival of a message header on one of the networks, the receiver must check whether it contains the correct sequence number. If the sequence number in the header matches the receiving sequence number for that MIN connection, then when

the associated user message arrives, it is passed to the user in his buffer. In either case, the size of the message is returned to the user as the return code to the **receive** call. If it is larger than the size of the buffer, then the user will know that the **receive** attempt failed, and how big of a buffer he needs.

Alternatively, the sequence number in the header can be greater than the receiving sequence number for that MIN connection. This implies that this message has arrived out of sequence, ahead of another message which was actually transmitted before this one was. The out of sequence message should be buffered, until the preceding message or messages have arrived and have been consumed by the user. Also, since messages sent on the same network cannot arrive out of order (guaranteed by the Transport service on each network), we may as well temporarily ignore the network on which the out of sequence message arrived, to decrease the chances of having to deal with more out of sequence messages. This network can be "re-enabled" after its buffered out of sequence message has been delivered to the user. If all the networks for the MIN connection have thus been disabled, it means that a message has been lost, and the connection must then be closed.

Another possibility is that the sequence number in the message is less than the receiving sequence number. This can be caused by an undetected network error which corrupted the sequence number in the MIN header, or it may imply the arrival of a duplicate message. In either case the dubious message should be discarded. Although

the Transport services for each network is supposed to provide reliable end-to-end communication, the nature of MIN's transmission algorithm makes duplicate messages possible. Suppose the Transport services of the chosen network successfully delivers the user message, but does not manage to get an acknowledgement from the Transport layer at the destination host. It will thus report the transmission attempt as having failed, even though it has been received by the destination Transport layer (there is no way to know that the acknowledgement was lost, since there is no acknowledgement of the acknowledgement). The MIN **send** primitive will then assume a failure on that network (which may actually be the case), and re-transmit on another network. When this message arrives via the second network and is passed to MIN, a duplicate message situation will have resulted. However, it is not a serious consequence, as the duplicate message can be simply discarded, and message reception can continue as normal.

One other possible scenario is that the MIN header is successfully sent, but the user message is not. In this case, the sender should re-send both the header and the message on another network. This is done in order to preserve the arrival sequence of header, message, header, message, etc. on each network. The fact that the transmission failure occurred on trying to send the message can be detected by the receiver upon the arrival of a duplicate header. That is, a header with the same sequence number has already arrived on another network. In this case, the first network to have delivered this header should be considered to have failed, and that network should be disassociated from this

MIN connection. The MIN software on the sender's side does not re-transmit either the header or the message on the first network, since each of the underlying protocols layers has already made a number of re-trys before giving up and reporting a transmission failure. The algorithm for receiving is given below.

There is also the issue of sequence number wraparound. The domain of sequence numbers should thus be chosen to be large enough as to make this rare, but not so large that it incurs a lot of overhead in terms of the size of the MIN header. An 8 bit sequence number should suffice. The sequence number is initialized to 0 when the MIN connection is created, and incremented once for each user message.

```
MIN Receive :
    Repeat
        Wait for a header or message to arrive on any network
        If an error occurs in trying to receive the header or message
            Disassociate this network from this MIN connection
            If all networks for this MIN network have been disabled
                Then return error
        Else If it is a header
            If message size > user's buffer size
                Then return size of message
            Record sequence number and message size for that network
            If sequence number > reception sequence number
                Temporarily disable that network
            Else If sequence number = reception sequence number
                and another network has already received the same header
                Disassociate that other network from this MIN connection
                If all networks for this MIN network have been disabled
                    Then return error
        Else /* it is a message */
            If recorded sequence number < reception sequence number
                Discard the duplicate message
            Else /* message is in-sequence */
                Put message into user's buffer
                Update reception sequence number
                Return size of message
```

Figure 2.15: Message Reception

Chapter 3

Implementation

This chapter deals with an implementation of the MIN software at the University of B.C. First the hardware and software environment is described. Next the configuration of the software, i.e. the user interface, is discussed. Following that is a brief report on the supporting system software used by MIN. Explanations on how this supporting software is utilized for connection and establishment and message exchange are included. Also contained in this chapter are sections dealing with how network loads are monitored and how path selection for message transmission is made.

It should be noted that the purpose of this implementation was mainly to obtain some measurement results for evaluating the practicality of the MIN concept. The fact that it was never intended for general use significantly influenced how certain aspects of the software was implemented. Evidence of this will appear in some of the upcoming sections.

3.1 Environment

The hardware environment for our implementation consists of a Vax 11/750, a Vax 11/780, and 10 Sun-2 workstations, interconnected by a 10 Mb/s Ethernet using 3COM interfaces [3COM82]. One of the Suns, named "ubc-dsrg", is also connected to a 10 Mb/s Toltec Cambridge Ring [Toltec]. Since the dsrg Sun is the only machine connected to more than one network, most of the testing and performance measurements (given in chapter 4), were performed on this workstation.

The Vaxes and Suns are connected to a number of terminals through a Develcon Data Switch. There are also dial-up lines, and remote logins are also possible through the Ethernet. All together, about 40 users may be simultaneously served. Unfortunately, though, this is hardly sufficient to load down the network.

All of the machines operate the 4.2 BSD Unix system, which provides local and remote transport level inter-process communication services in the form of *sockets*. These sockets are used by MIN as the interface to the system's supporting network protocols, for both the Ethernet and the Cambridge Ring. Before discussing how exactly the sockets are used, the IPC primitives offered by 4.2 BSD are first presented. Only a very brief description is given; much more detail can be found in [Leff83].

3.2 4.2 BSD IPC Primitives

When a process running under 4.2 BSD Unix wishes to communicate with another process, each of these processes must first create a *socket*. This is done using the primitive :

socket(domain, type, options)

The first parameter specifies the communication domain. If it is local (both processes reside on the same host), then the user use the constant AF_UNIX as the **domain** argument. If it is remote, the constant AF_INET (for inter-net) is used. The parameter **type** defines the type of socket to create. The types available are stream (specified by the constant STREAM), for use in connection-oriented IPC, and datagram (specified by the constant DGRAM), for connectionless IPC. The MIN software uses only stream sockets. The value returned by **socket** is a non-negative socket number.

For connection-oriented IPC, once each process has created a socket a logical connection must be made between them. To achieve this, one the processes must act as a client and make a connection attempt to the other process, which is acting as a server by passively waiting for the connection attempt. The client's attempt is carried out

using the primitive :

connect(socket, sin, sinsize)

The first parameter is the socket number returned by **socket**. The second parameter points to a structure containing information on the server process which the client wishes to connect to, and the host machine which the server resides on. The third parameter indicates the size of the **sin** structure. The host and server information is obtained through 2 system functions :

gethostbyname(hostname)

getservbyname(servername)

These functions allow users to refer to hosts and servers by symbollic names (Ascii strings). In 4.2 BSD there is a names service which enforces unique server names on a per-host basis, which is why both the host and server must be specified in the call to **connect**.

Before the client can connect to a process named **servername**, the server must

have already assumed that identity. This it does using the primitive :

bind(socket, sin, sinsize)

The **sin** and **sinsize** parameters are the same as those for **connect.** However, only the server information in the **sin** structure needs to be filled in, and not the host information. The server information is obtained using a call to **getservbyname.**

Once the server process has logically bound a socket to the symbollic name, it can then sit and wait for a connection attempt by a client. This is done using the primitive :

listen(socket)

This primitive only needs to be called once by the server. It marks the specified socket as "listening" for connection attempts. When one arrives, the server may accept the connection request using the primitive :

accept(socket, sin, sinsize)

The **socket** parameter is the number of the bound socket which the server is listen-

ing on. The **sin** and **sinsize** paramters are the same as those for **connect**. However, the **sin** structure is filled in by the system rather than the user. On return from the **accept** call it contains information on the client whose connection request was just accepted.

A connection is now established between the server and client. The **accept** primitive creates a new socket for the server which is used for subsequent data transfer on that connection. The original socket is retained for listening for additional connection attempts by other clients.

To send data through a connected socket, a process uses the primitive :

write(socket, address, size)

This transfers **size** bytes of data located at **address** through **socket** to the socket at the other end of the connection. The **write** primitive returns the number of bytes successfully written. To retrieve the data, the other process uses the primitive :

read(socket, buffer, size)

This attempts to get up to **size** bytes of data out of **socket** and put them into **buffer**. The **read** primitive returns the number of bytes successfully read.

Although data transfer on 4.2 stream sockets behaves more or less like standard Unix I/O (i.e. a boundaryless byte-stream), there are some differences. Suppose two processes have established a connection of stream sockets, and one process executes :

```
read( socket, buffer, 96 )
```

while the other process executes

```
write( socket, address, 64 )
write( socket, address, 32 )
```

The result returned to the receiving process for its **read** call may be different depending on the amount of time which passes between the 2 **write** calls made by the sender. If a long time elapses in between, then the read call returns 64, the number of bytes sent by the sender on its first **write**. That is, only the first chunk of data is delivered to the receiver. If there is a very brief interval, then all 96 bytes are delivered. This is because the 4.2 BSD software on the receiving side only waits a finite amount of time after the first block of data arrives for more data to arrive. If no further data arrives within that period, then it simply delivers whatever it has gotten so far. However, this characteristic of 4.2 BSD is not of great consequence as far as the MIN

software is concerned.

When a process has finished using a socket, it can be removed using the primitive :

close(socket)

As pointed out in chapter 2, it is crucial to the MIN software to be able to wait for more than one event simultaneously. The events of interest are the arrival of messages and connection requests. In 4.2 BSD there is a primitive which provides the ability to multiplex socket I/O :

select(maxsocket, readmask, writemask, exceptmask, options)

The parameters **readmask**, **writemask**, and **exceptmask**, are bit masks of socket numbers which the user is interested in using for receiving data, sending data, and transmission exceptions, respectively. The MIN software only needs to use **readmask**, so **writemask** and **exceptmask** will not be further discussed. The final parameter **options** is for specifying additional options. This is not used by MIN either.

Select is not a blocking function which does not return until one of the specified events (e.g. arrival of data on one of the sockets specified in **readmask**) occurs.

Rather, it takes an instantaneous check on the status of each of the specified sockets and reports on them. Thus to wait for an event, the user must repeatedly call **select**, in a polling fashion.

To check whether any data has arrived on a certain socket, the user must set the corresponding bit in argument **readmask** to 1. For example, to check the socket whose id is 2, bit 2 (where bit 0 is the least significant bit) in **readmask** should be set to 1 on entry to **select**. On exit, the same bit in **readmask** is set by the **select** function to indicate that socket's receiving status. If the bit is 1, it means that some data has arrived on that socket which has not been delivered to the process yet. A bit value of 0 on the other hand mean that there is no outstanding incoming data on that socket. Thus **readmask** acts as both an input and an output parameter.

More than one bit may be set in **readmask**, to allow multiple sockets to be examined for their status simultaneously. The limit depends on the host machine's word size (number of bits in **readmask**) and maximum number of sockets which may be open at the same time. On our Sun workstation the limit is 32, which is the word size.

The **maxsocket** parameter is used to indicate the largest socket number specified in **readmask**, **writemask**, or **exceptmask**. In our case only **readmask** is relevent. That is, **maxsocket** should be set to the number of the most significant "on" bit in **readmask**, plus one. For example, if the highest socket number selected is 6, so that bit 6 in **readmask** is set to 1, then **maxsocket** should be given the value 7.

The **select** function can be used to check for the arrival of a connection request as well as data, if the socket is marked as "listening". In either case the corresponding bit in **readmask** has the value 1 on exit from **select**. Thus a connection request could be viewed as a special type of data.

3.3 Supporting Network Software

Communication over the Ethernet is provided directly by 4.2 BSD, which runs TCP/IP as the default transport protocol for internet stream sockets. At U.B.C. there is another transport level protocol available, the locally developed LNTP (Local Network Transport Protocol) [Chan84]. Although LNTP is designed especially for LANs and is therefore much more efficient in our environment, TCP/IP was chosen for our implementation because it is much more widely used. Switching to LNTP or any other transport protocol would be easily done as it merely involves the modification of one of the parameters supplied to the 4.2 BSD IPC function **connect** (see 3.2).

Access to the Cambridge Ring is provided by a software package developed by another graduate student at U.B.C., L. Chan [Chan85]. This software also provides transport level communication, using the BSP and BB protocols. Unlike the TCP/IP software, which sits passively in the 4.2 BSD kernel, the BSP software is structured as an active server process. The BSP server is interfaced through 4.2 local IPC, using

Unix domain sockets.

The communication provided by the BSP server is connection-oriented, so that a client/server model is used for connection establishment. This begins with the user server process sending an **accept** request message to the BSP server. This message contains a BSP connection id which identifies the user to the BSP server. Unfortunately the current state of the BSP software is such that these connection ids must be determined by the user himself, rather than given out by the BSP server. That is, the user must choose an id somehow and make sure that it is not used by anyone else. How the MIN software deals with this is discussed in Section 3.6.

After the user server process has established contact with the BSP server, then the user client process can send a **connect** request message to the BSP server. This message should contain a connection id for the client itself, and the connection id used by the server which the client wishes to connect to. If the server connection id is incorrect, or the **connect** request message is received by the BSP server before the **accept** request message, then an error reply message is returned to the user client process.

To send a message over the Cambridge Ring, a user process must first send a **send** request message to the BSP server, which contains the destination connection id, and the size of the message. The user process then sends the message itself to the BSP server. A reply message is returned by the BSP server to indicate the results of the

transmission attempt.

To receive a message over the Ring, a **receive** request message is sent to the BSP server. This request message contains the user process's connection id, and the size of the expected message. The user process must know ahead of time how large the arriving message will be. If it turns out to be of a different size, then problems will develop. Fortunately this characteristic of the BSP server interface does not pose too great a difficulty for the MIN software (see Section 3.7).

When a BSP connection is no longer needed, it may be removed by sending a **close** request message to the BSP server, and then closing the local socket.

3.4 Software Configuration

The MIN software is intended to provide communication services to all user processes, as part of the host system. Therefore it should reside in system space, as a re-entrant shared module. However, in our implementation it was decided not to place the MIN software in the host system, in order to disturb users as little as possible. Instead the MIN software is configured as a linkable library of routines, so that each user program contains its own copy of the code. Another motivation for this was that the BSP server is itself a user process, accessed through system IPC services. In our environment the 4.2 BSD IPC functions form the Application layer. If the implementation were intended

for actual use rather than experimental purposes, then the MIN software would need to be incorporated into the system.

The MIN library contains the 5 primitives described in Section 2.4. A synopsis of each appears below. These functions are implemented in C, and are intended for a C caller. The prefix **min_** is added to their names in order to avoid confusion with the 4.2 BSD IPC routines.

```
int min_accept( servername, networks )
        char *servername;
        int networks;

int min_connect( hostname, servername, networks )
        char *hostname;
        char *servername;
        int networks;

int min_send( MINid, message, messagesize, networks )
        int MINid;
        char *message;
        int messagesize;
        int networks;
```

```
int min_receive( MINid, buffer, buffersize )
        int MINid;
        char *buffer;
        int buffersize;

int min_close( MINid )
        int MINid;
```

The **networks** parameter for **min_accept** and **min_connect** is a bit mask of network ids. Its format is similar to that of the **readmask** parameter for the 4.2 BSD function **select** (see Section 3.2), which is a bit mask of socket numbers. Each network in the MIN environment is assigned a small integer id, starting from 0 as the lowest id. To specify a network using **networks**, the bit which corresponds to that network's id is set to 1. For example, to specify the network whose id is 4, use :

```
networks = 1 << 4
```

In our implementation environment the Ethernet is assigned the id 0 while the Cam-

bridge Ring is given id 1. These ids are also defined as the C constants MIN_ETHERNET and MIN_CAMBRING, respectively, in the header file **min.h**. Thus to specify the Cambridge Ring for example, use :

networks = 1 << MIN_CAMBRING

By using **networks** as a bit mask, more than 1 network can be specified using a single value. This is achieved by setting the bits corresponding to each desired network to 1, using a logical OR operation. For example, to select both the Ethernet and the Cambridge Ring, use :

networks = (1 << MIN_ETHERNET) | (1 << MIN_CAMBRING)

If the user does not care which networks are used for the MIN connection, the **networks** parameter should be set to 0, or the defined constant MIN_ANYNETS. This symbol is also defined in the file **min.h**, along with a number of other items which will be mentioned in the upcoming sections.

The **servername** parameter is a pointer to a null-terminated Ascii string which symbolically identifies the server process. This string must be a name which is understood by 4.2 BSD function **getservbyname** (see 3.2). Two names were created for the

purposes of developing MIN : "kc_test1" and "kc_test2". The **hostname** parameter also points to an Ascii string. It identifies the host which the server resides on. It must be a name understood by the 4.2 BSD function **gethostbyname**, for instance "ubc-dsrg" for the dsrg Sun workstation.

The **min_accept** and **min_connect** primitives return a non-negative MIN connection id when a MIN connection is successfully established. This MINid is used for subsequent calls to **min_send**, **min_receive**, and **min_close**. If an error occurs, then a negative error code is returned instead.

The **min_send** primitive returns 0 if the message located at **message**, of size **messagesize** bytes, is successfully sent. A negative error code is returned otherwise. The **min_receive** primitive returns a positive value equal to the size of the next received message. If the size of the user's buffer, **buffersize**, is large enough, then **min_receive** copies the message into **buffer**. If the user's buffer is not big enough, then it is left undisturbed. The user will know that a message has arrived but has not been passed to him, by the fact that the return code from **min_receive** is larger than his buffer size. Another call to **min_receive**, with a large enough buffer, would have to be made in order to retrieve that message. If some reception error has occurred, then **min_receive** returns a negative error code. One final possibility is that the process at the other side of the MIN connection has closed his end of the connection, so that no more messages will arrive on it. In this case **min_receive** returns 0.

The **min_close** primitive only requires a MIN connection id as a parameter. It returns 0 if the connection is successfully closed, or a negative value if there is an error. The only possible error is that **MINid** is invalid.

3.5 Host Reachability

Our implementation of MIN uses a static scheme for recording host reachability information. That is, only the physical connection of the machines to the networks is stored. No attempt is made to keep track of the current up/down status of each host and network.

The logical structure for storing the host reachability information is the linked list of linked lists structure described in Section 2.5. It is implemented as an Ascii text file, with each line containing the names of the hosts which are connected to that network. The first line is used for network 0 (Ethernet) and the second line is used for network 1 (Cambridge Ring). The file is kept in Ascii instead of binary form so that it can be more easily read and modified. This file should be updated whenever the physical connectivity of hosts and networks in the MIN domain changes.

Each of the host names in this file are those used for the local networking environment. That is, these names are understood by the system function **gethostbyname**. The host names on each line are separated by one or more blanks or any other white-

space characters (tabs, etc.). When the **min_connect** primitive is called by a user, the MIN software searches each line of this file for the argument host name, to determine which networks that host is connected to. If the name is not found on any line, then it is an invalid name and **min_connect** returns an error code. Otherwise, if the user has specified a set of networks to use, then the MIN software must verify that the argument host name is found on the appropriate lines of the file. If it is missing from any of the specified network's corresponding lines, an error code is returned.

The same check on the **networks** parameter is made by the **min_accept** primitive. However, since the user is not required to submit the local host name as an argument to **min_accept**, the MIN software must either have it hard-wired into the code or determine it dynamically from the system. Since hardwiring the host name would require a different version of MIN on every machine, getting it at run-time is a much better idea. On 4.2 BSD there is a system function which provides this :

gethostname(hostnamebuffer)

This function returns the name of the host in the argument buffer, as an Ascii string which is understood by the system function **gethostbyname**. It can therefore be used as the target string in searching each line of the host reachability file.

There is the possibility that the file may be modified while it is being examined by

the MIN software. In order to take the changes into account as soon as possible, the MIN software reads in each of the file every time it wants to perform a search for a host name, rather than reading in the entire file only once at startup time and then performing all subsequent searches on the in-memory copy. Although the alternative method of keeping a copy of the file in memory may save a little bit of I/O time, the results might not always be correct. For instance, if the application is a system deamon which is up for a long time, then it would not discover the changes in the host reachability file at all.

3.6 Connection Establishment

To establish a MIN connection between 2 processes, the MIN software must do the following :

1. establish a connection on each of the networks on which the 2 processes can reach each other, and which were selected by both the client and server in their respective **networks** argument
2. ensure that both sides have the same view of which networks are being used for that MIN connection
3. inform each side of the connection of the MINid assigned to that MIN connection at the other side, so that message exchange can successfully take place

The final step is not necessary in our implementation since the supporting network software is connection-oriented. This means that we only need to know the socket

number on our side of a connection when we wish to transmit a message. Since a logical connection has been established between that socket and the destination process' socket, the supporting software will deliver the message to the appropriate destinattion socket, and hence to the correct destination process.

On the client's side, the MIN software makes a connection attempt on each of the networks selected for this MIN connection. These are the networks found in the host reachability file (see 3.5) which were selected by the user in the **networks** argument passed to **min_connect**. For the Ethernet, the connection attempt is made by creating a socket and then calling the 4.2 BSD routine **connect** (described in 3.2). If it is successful, then the socket number is recorded in a data structure which maintains the status of each MIN connection. The organization of this structure is shown below, in C programming language syntax.

The structure member **type** is a sub-structure which details which networks are utilized for this MIN connection. Its format is shown below.

The structure member **many** indicates whether only one network is used, or more than one. If only one, then the union **t** contains the id of that lone network. If more than one network is used, then the union **t** holds a bit mask of all the network ids used. This bit mask is in essentially the same format as the **networks** parameter for **min_connect**, **min_accept**, and **min_send**. By considering a MIN connection which only involves a single network to be a special case, it allows messages to be sent without

```
typedef struct
{
        TYPE type;
        int smask;
        int maxs;
        int rmask;
        int sseq;
        int rseq;
        PORT port[MIN_MAXNET];

} CONNECTION;
```

Figure 3.1: CONNECTION Structure

going through the path selection algorithm first (discussed in 3.9), and messages to be received without polling all the sockets (discussed in 3.9).

The members **smask**, **maxs**, and **rmask** of the CONNECTION structure are used for message reception, but only when the MIN connection involves more than one network. The members **sseq** and **rseq** are message sequence numbers used for sending and receiving, respectively, on this MIN connection. Message exchange is discussed in detail in Section 3.9.

In addition to storing data which relate to the MIN connection as a whole, some information must be kept for each of the networks used for that MIN connection. This is kept in sub-structures, one per network connection. Each of these is termed a PORT.

```
typedef struct
{
        BOOLEAN many;
        union;

                int net;
                int nmask;
         t;

} TYPE;
```

Figure 3.2: TYPE Structure

Each PORT structure contains a socket number used for communicating over that network. For the Ethernet this would be a socket which is connected directly to the process on the other end of the MIN connection. For the Cambridge Ring this socket would be connected to the BSP server. As mentioned in Section 2.9, a MIN header logically prepended to each user message, and although the header and message are not sent together as a single unit, they are always launched on the same network, with the message immediately following the header. Thus the arrival sequence of items on each network is header, message, header, message, and so on. The PORT structure must therefore keep track of whether it is a header or a user message which is due to arrive next on that network. This is done using the boolean structure member **rdata**.

```
typedef struct
{
        int sock;
        BOOLEAN rdata;
        HEADER rhdr;

} PORT;
```

Figure 3.3: PORT Structure

When it is a message which is due to arrive next, **rdata** has the value TRUE. When it is a header which is due to arrive next, **rdata** has the value FALSE. The PORT structure also contains a buffer for storing the latest header received, which contains relevent information on its associated message, such as its size. The format of the HEADER structure and its usage is discussed in Section 3.9.

When the **min_connect** primitive makes a successful connection on a network, the socket number is recorded in that network's PORT structure. If this MIN connection involves more than one network, then the MIN software also sets the bit corresponding to the socket used for this network in the **smask** member of the CONNECTION struture. As mentioned earlier, **smask** is used for message reception, which is discussed in Section 3.9. If on the other hand the connection attempt failed, then the **min_connect** primitive must dissociate that network from this MIN connection, so that this network

would not be considered for subsequent message transmission and reception. This is done by clearing the bit corresponding to that network's id in the TYPE sub-structure within the CONNECTION structure. A connection attempt may fail due to a number of reasons. One possibility is that there is a problem with the network or the destination host. Another possible cause is that the server is not present or is not listening for connection attempts on that network. If connection is not successfully established on any network, then **min_connect** returns an error code.

Once a connection attempt has been made on each of the networks selected for this MIN connection, and the results appropriately recorded, then the **min_connect** primitive sends the connection information in the TYPE sub-structure over one of the networks on which connection is established. This action serves two purposes. First, it tells the process at the other end of the connection which networks the process at this end thinks connection is established on. Secondly, it lets the server know that the client has completed all his connection attempts and is now ready for message exchange. The latter is useful in case the server is listening for connection attempts on more networks than the client is trying to connect on. For example, suppose that the server is listening on both the Ethernet and the Cambridge Ring, but the client only wants to connect on the Ethernet. When connection has been established on the Ethernet and the TYPE information arrives, the server will then know that this client only wants to connect on the Ethernet. It can therefore stop listening for connection

attempts on the Cambridge Ring. Since a client's connection attempt blocks him from further execution until the server has received and accepted it, it is not possible for the TYPE information to arrive at the server until the client has completed all of his connection attempts. Therefore there is no possibility of the TYPE information arriving too soon and causing an error.

However, if there is more than one client attempting to connect on more than one network around the same time, then problems may arise if their connection attempts are interleaved. For example, suppose there are two clients *client*1 and *client*2, both wishing to connect to the server on both the Ethernet and Cambridge Ring. Suppose further that *client*1*'s* Ethernet attempt has already arrived and has been accepted, when *client*2*'s* Cambridge attempt arrives. If the MIN software does not realize that this attempt is from a different client than the one accepted on the Ethernet, then it would incorrectly form a MIN connection from the Ethernet connection to *client*1 and the Cambridge Ring connection to *client*2. Thus some client identification information must be retained when a connection attempt is accepted. Fortunately this information is made available by the 4.2 BSD routine **accept**, so that **min_connect** does not have to send it itself.

Another scheme which was considered as a possible solution to the above problem is to order all the networks in our environment so that clients always runs through the networks for their connection attempts in the same order. The most obvious ordering

is by network ids, so that in our implementation a connection attempt would always be made on the Ethernet before the Cambridge Ring. This scheme works very well for the scenario described above. Since the server has already accepted *client1's* connection on the Ethernet, it ignores *client2's* connection attempt on the Ethernet. Thus *client2* is blocked, and it cannot make a connection attempt on the Cambridge Ring and have it arrive before *client1's* Cambridge Ring connection attempt. Unfortunately, though, this method does not always work. A counter-example is easily formed by modifying the above situation so that *client2* is only attempting to connect on the Cambridge Ring.

For the server, the MIN software's task is different depending on whether the MIN connection involves more than one network or not. If only one network is used, then the **min_accept** primitive merely has to create a socket, and then wait for a connection attempt from the client.

If more than one network is involved, then the server must create a socket for each of these networks. It then polls all the sockets at the same time for the arrival of connection attempts (on the Cambridge Ring the server must first send an **accept** request message to the BSP server). When a connection attempt arrives on one of the networks, the MIN software accepts it. If it arrived on the Ethernet, then the new socket number returned by the 4.2 BSD function **accept** must be stored into the PORT sub-structure. The MIN software then records the fact that connection has been

established on this network by clearing the bit corresponding to the socket used for this network in the **rmask** member of the CONNECTION structure (**rmask** is initialized to the value of **smask** after all the sockets are created). However, **min_accept** does not ignore this network but continues to listen to all the networks selected for this MIN connection. Later, when the 4.2 BSD function **select** reports that something has arrived on an already connected network, the MIN software will know that it is not a connection attempt which has arrived. but rather the TYPE information sent by the client to indicate the end of the connection establishment phase. The **min_accept** primitive can then stop listening for connection attempts on the yet unconnected networks if there are any, and dissociate those networks from this MIN connection by clearing the corresponding bits in its CONNECTION structure's TYPE sub-structure.

A check should be made that the local TYPE information matches the contents of the TYPE sent by the client. In the unlikely event that they disagree, all network connections just established for this MIN connection are removed and **min_accept** returns an error result.

3.7 Path Selection

When the user wants to send a message over a MIN connection which uses more than one supporting network, a path selection must be made. If the user has specified

a network or a set of candidate networks using the **networks** parameter passed to **min_send**, then only these networks are considered, if they are valid for this MIN connection. The validity of the specified networks are easily determined using the TYPE information in the CONNECTION structure. If any of the specified networks are invalid, i.e. connection was not actually established on those networks or the destination host is not reachable on those networks, then those networks are not used for the path selection algorithm. If the user did not specify any networks using the **networks** parameter, which would be the usual case, then the MIN software will choose from the networks indicated in the TYPE sub-structure.

The path selection algorithm used by our implementation considers two factors : message size and network loads. A prioritized scheme, as described in Section 2.7, is used, with message size treated as the more important of the two factors. Since the Cambridge Ring is more suitable than the Ethernet for sending small messages, the Cambridge Ring is chosen for sending all messages <= 32 bytes. On the other hand, the Cambridge Ring is not nearly as efficient as the Ethernet for sending large messages. Thus the Ethernet is chosen for sending all messages >= 512 bytes. These size boundaries were not mathematically derived to be the optimum; their selection was based on the fact that the Cambridge Ring's Basic Block protocol only puts 2 bytes of actual data in each mini-packet. Also, the Ethernet always pads packets to a minimum size of 128 bytes. These size boundaries are reasonable for our purposes, but

for an implementation which is intended for general use, more careful analysis would need to be done.

For messages with sizes between 32 and 512, the path selection is based on the current load on the two networks. The **min_send** primitive chooses the network with the lighter load to send the message on. How the network loads are estimated and accessed by the MIN software is discussed in the next section.

3.8 Load Monitoring

Our implementation of the MIN software uses the length of the outgoing queue at the lowest protocol layer as an estimate on the network load. The length of the queue is measured in terms of the number of packets in the queue, rather than the total size of the data contained in the packets. Although this value only reflects the local load, it is a reasonable measure to use since the length of the transmission queue has a direct effect on the delay before the message is actually transmitted, and the purpose of monitoring the network loads is to try to get the message launched as quickly as possible.

In order to store the load information, the device drivers for the Ethernet and Cambridge Ring were modified to each keep a counter for the length of its outgoing queue. They were also changed to accept a new type of I/O control request, for inquiring on

the current network load (length of the transmission queue). Thus the load information for a network is obtained using the system function **ioctl**, as follows :

```
ioctl( dd, GETLOAD, &load );
```

The argument **dd** is a device descriptor number, obtained from an earlier call to **open** the device (Ethernet or Cambridge Ring). The argument **GETLOAD** is a defined constant which uses the system-defined macro **_IOCTL** to construct a packed 32 bit value which specifies that this is a request to get the current network load. The final argument is the address of an integer variable **load**, into which the modified device driver will place the network load value.

To avoid direct access of the physical layer device drivers by the MIN software, a Network Monitor process was built which made the **ioctl** calls to get the network loads information and forwarded these to the MIN software through Unix domain sockets. However, as one would expect this proved to be much too expensive, as a request and reply would need to be exchanged each time a user message to be transmitted requires a path selection based on network loads. Thus the **ioctl** calls have been moved into the MIN software, and are made by the **min_send** primitive instead.

3.9 Message Exchange

The **min_send** primitive was relatively straightforward to implement. When the MIN connection only involves a single network, all user messages are sent on that one network. The MIN software sends a header ahead of each user message. The header structure is organized as follows :

```
typedef struct
{
        unsigned seq;
        unsigned size;

} HEADER;
```

Figure 3.4: HEADER Structure

The structure member **seq** is the sequence number for that message. This is the sending sequence number for this MIN connection, stored in the **sseq** member of the CONNECTION structure. The sequence number is initialized to zero when the MIN connection is created.

When a message is to be sent on a MIN connection which involves more than one network, the MIN software must go through the path selection algorithm described in Section 3.7. The message is then sent on the selected network, preceded by the

header described above. If an error occurs in attempting to send either the header or the message, the MIN software assumes that this network has failed, and dissociates it from this MIN connection by clearing the corresponding bit in the CONNECTION structure's TYPE sub-structure. The **min_send** primitive must then select one the remaining networks for this MIN connection for re-transmission. On the re-try both the header and the message are sent, even if the header was successfully sent on the first try and the error had occurred in trying to send the actual message. This is done in order to preserve the arrival order of header, message, header, message, and so on for each network, which simplifies the task of the MIN software for receiving messages. The re-transmission process continues until either it is successful or a failure has occurred on every network for this MIN connection, in which case the **min_send** primitive returns an error result and removes this MIN connection altogether.

Although our implementation could have been simplified somewhat by taking into consideration the fact that we are only dealing with two networks, we instead constructed our software to handle the general case of n networks. By doing so we lend greater validity to the measurement results obtained, and allow the simulation and testing of an n-network situation.

The task of receiving a message on a MIN connection which only involves a single network is also relatively simple. The MIN software primitive first waits for a header to arrive on that network, then checks the size of the following message indicated in

the received header against the size of the buffer passed by the user to **min_receive**. If the user's buffer is not big enough, then **min_receive** immediately returns the size of the expected message, to let the user know that he needs a larger buffer. When **min_receive** is called again, the MIN software will know that this is a reception re-try on the part of the user, by examining the **rdata** flag in the PORT structure for this network (this flag was toggled previously upon receiving the header). Again the user's buffer size is checked for adequacy. If it is big enough this time, then **min_receive** waits for the message to arrive, and puts it into the user's buffer. The size of the message is returned. If an error occurs in trying to receive either the header or the message, the **min_receive** primitive returns an error result and assumes that the network has failed. Since it is the only network for this MIN connection, the MIN connection is removed. Also, even though the underlying network software is supposed to provide reliable service, the sequence number in the received header is checked against this MIN connection's reception sequence number (stored in the **rseq** member of the CONNECTION structure). If the sequence numbers do not match, then something has gone wrong with the supporting software for this network, and the MIN connection must be removed.

The only case for which the MIN software's task is relatively complex is that of receiving a message on a MIN connection which involves more than one network. Here, the **min_receive** primitive repeatedly polls the status of all the networks used for this

MIN connection using the 4.2 BSD routine **select**, as described in Section 3.2, until something arrives on one of the networks. To determine whether it is a header or data which has arrived, the **rdata** flag in the PORT structure for this network is consulted. If it is a header, then it is received using the 4.2 BSD routine **read** and placed in the PORT structure. The MIN software then toggles the **rdata** flag to indicate that it is the message and not a header which is due to arrive next on this network. Next, the sequence number in the received header is examined, and compared against the the reception sequence number for this MIN connection, stored in the **rseq** member of the CONNECTION structure. If they differ, then it is an out-of-sequence header (and message). Since each network is assumed to preserve the order of messages sent on it, the MIN software can therefore temporarily ignore this network, and only poll the other networks in search of the in-sequence message. This is done by clearing the bit in the **rmask** member of the PORT structure which corresponds to the socket number used for communicating on this network. Since **rmask** is used as an argument to **select**, this effectively excludes that network in subsequent polling.

Since sequence numbers wrap around, it is not possible to recognize the re-appearance of an already received header by virtue of its sequence number being "less than" the expected sequence number. However, if a header is received in-sequence, which has already been received on another network, then we can conclude that the network on which that header was first received has failed, and the sender is now using the other

network for re-transmission. This conclusion is consistent with how transmission errors are handled by **min_send**, as described above.

After receiving an in-sequence header, the MIN software then checks whether the user's buffer is large enough. If not, then **min_receive** immediately returns the size of the incoming message to indicate buffer size deficiency to the user. If the buffer is large enough, the MIN software continues polling until the message has arrived on that network. Its contents is then obtained using the 4.2 BSD function **read** and placed into the user's buffer.

If an error occurs in trying to receive either the header or a message on a network, then that network is dissociated from the MIN connection by clearing the corresponding bit in the CONNECTION structure's TYPE sub-structure. Also, to prevent further polling of this network, the bit corresponding to the socket number used for this network in the **smask** member of the CONNECTION structure is cleared. The **rmask** member is set to the value of **smask** after each message is successfully received, so that all networks will be polled again on the next invocation of **min_receive**.

Chapter 4

Measurement

This chapter reports the results of some measurements and a simulation. The measurements were taken in order to determine the overhead incurred by the MIN software. This is discussed in the first section. The second section deals with a simulation study set up to provide some idea on the type of conditions under which the use of multiple networks with MIN would improve communication performance. There are a number of reasons for using a simulation instead of taking actual measurements even though we have implemented the MIN software in an environment which contains an Ethernet and a Cambridge Ring. These will be explained in the second section, along with the simulation results.

4.1 Overhead of MIN

By using MIN, a process incurs the overhead of having to go through an additional layer of software. In order to determine how much of an overhead MIN is, two sets of measurements were taken. Both sets consisted of a number of test runs, in each of which a client process first connects to a server process then sends him a large amount of data in fixed-size messages. The time required to transfer the data is measured at the server, starting from the moment the client's connection request is accepted and ending when the last message is received. The client and server in the first set of measurements use 4.2 BSD IPC services over the Ethernet. The client and server in the second set of measurements also communicate over the Ethernet, but they use the MIN primitives we implemented on top of the 4.2 BSD services. Since the only difference between the 2 sets of measurements is that MIN is used for the second set but not for the first, by comparing the results we should be able to determine the overhead incurred by MIN.

In each set of measurements two factors were independently varied : the total amount of data transferred and the size of each message. This was done in order to determine what effects if any these factors have on performance. For each combination of (total data, message size), 10 measurement runs were made without MIN and 10 made with MIN. The average total transfer time for each group of 10 runs were

calculated. These results are shown in Table 4.1.

These measurements were taken late at night, when there were no other users on the machines. The data transfer was remote. That is, the client and server resided on different SUN workstations. In both sets of measurements only one network, the Ethernet, was used. It was planned to perform the same measurements on the Cambridge Ring. However they were not done because of hardware problems and the unreliable nature of the local implementation of the Cambridge Ring protocols.

From the results in Table 4.1 one observes that the percentage difference between using and not using MIN decreased with the message size. This behaviour is intuitively reasonable. The overhead of MIN is composed of two portions. One is the CPU cycles expended in executing the MIN software. The other is the work required for the supporting network protocols to send the MIN message header. The CPU time used for MIN should be constant regardless of the message size since there is no data copying of the message's contents within MIN. On the other hand the relative size of the fixed-length MIN header decreases as the size of the message increases. This explains the decreasing nature of the percentage overhead of MIN. Therefore if the results were expressed on a per-message basis rather than for the overall data transfer time, the MIN overhead should be basically constant. It should be unaffected by either the message size or the total amount of data transferred. These result are shown in Table 4.2.

One other observation which may be made from this table is that there appears to be a performance peak when the message size is 1024 bytes and MIN is not used. That is, the tendency for the time to send a message to increase as the size of the message increases does not hold when the message size is 1024 bytes. A similar observation has been reported by Cabrera [Cabr85], whose investigation uncovered the cause to be the buffer management technique in 4.2 BSD's implementation of TCP/IP. When the message size is exactly 1024 bytes, internal copying of the message data within the 4.2 networking software is done by augmenting a counter. When the message size is not 1024 bytes, physical copying is done, which is a more expensive operation.

However, this peak phenomenon does not occur in the case where MIN is used. Without MIN the time per message is less for a 1024 byte message than for a 512 byte message, but with MIN the results are reversed. This is because the gain in 4.2 BSD performance for a 1024 byte message (about 2 msec) is over-shadowed by the MIN overhead (about 5 msec).

In these first two sets of measurements, a client sends continuously to a server with no return data from the server. Since both the 4.2 BSD IPC services and the MIN primitives perform sends in a non-blocking manner, the sender can keep sending ahead until the buffers are exhausted in one of the supporting protocol layers in either the client's or the server's host. With the MIN software sending a header for every user message, the buffers will fill up sooner with MIN than without. Therefore this

particular scenario can be taken as the worst case for the MIN overhead. The opposite situation would be one in which the message transmission alternates, with each client-sent message followed by a server-sent message. There would be no sending ahead at all. This scenario was used to obtain two additional sets of measurements. The total data sent by the client as well as the message size were set to the same values as those used for the first two sets of measurements. The server's reply is of the same size as the client's message. The per-message results are shown in Table 4.3.

In this table we observe that there is no performance peak at a message size of 1024 bytes either with or without MIN. It was explained earlier that physical data copying is unnecessary inside 4.2 BSD when the message size is 1024 bytes. In addition to the time saved by not doing data copying, more buffer space is available since only one copy of the data exists instead of two or more. Therefore more sending ahead can be done, further benefiting the one-way client send to server situation used for the first two sets of measurements. However, in the second two sets of measurements there is no sending ahead at all, so that there is no gain from the increased availability of buffer space. This would explain the lack of a performance peak at a message size of 1024 bytes in the results of Table 4.3.

4.2 When Multiple LANs May Improve Performance

As stated earlier one of the possible benefits of using multiple networks is improved performance. That is, by choosing the network which we think has the lightest load for sending each message, the total time required may be less than if we were to use a single network only. As well, the total network throughput rate will increase with more than one physical communication medium. However, the time saved by choosing a lightly-loaded network must be greater than the processing overhead incurred in making the path selection. In order to determine the network load conditions under which using multiple networks through MIN would out-perform using a single network without MIN, a simulation study was performed.

There are two main reasons for using a simulation instead of taking actual measurements despite the fact we have implemented MIN in an environment which contains an Ethernet and a Cambridge Ring. First, we lack the hardware to sufficiently load down the networks. Second, it is easier to vary the network loads using a simulation. Although a simulation suffers the disadvantage of possibly not taking all the necessary factors into account, it is sufficient for our purposes since we are not after exact results.

The simulated scenario is similar to that for the first set of measurements reported in Section 4.1, in that it involves a client sending continuously to a server. However, here the comparison is between using a single network without MIN and using two

networks with MIN. Also, the simulation focuses on what goes on at the client's host, and disregards the server. It divides the time required to send a message into 3 parts. The first part is the processing delay for the client's message to filter down through the various protocol layers and queue up at the physical layer. This is termed the protocol delay. The second part is the amount of time the message spends before it reaches the head of the queue : the queueing delay. The third part is the time needed for the message at the head of the queue to be successfully sent : the launch delay. This launch delay includes the time to attain the network, and any re-transmissions required due to error. Re-transmissions due to error is a random quantity while the time needed to access the network can be interpreted as a measure of the network's global load.

The protocol delay in filtering through the various layers depends on the particular protocols involved. For simplicity we assume it to be the same for both networks, although the simulation program is set up so that it can be individually specified for each network.

The queueing delay is handled by actually implementing a queue for each network in the simulation program. That is, after the protocol delay when the message enters the physical layer, it is added to the end of the FIFO queue. If it is the only message in the queue, then an attempt is made to launch it (see launch delay below). Also, a maximum queue length is defined. If the queue is now full, then the user process is

blocked and not allowed to generate more outgoing messages until the message at the head of the queue is successfully sent. This will be described in more detail shortly, when the event sequences are discussed. When the queue is not full then the user process is allowed to send ahead, by generating another message as soon as the first message is put into the queue.

In most cases, a user message is divided into one or more packets or frames by the time it reaches the physical layer. The number of packets a message is divided into depends on the size of the message. However, by assuming the messages to be of fixed size, we can define the maximum queue length in terms of messages instead.

The launch delay is how long it takes for the message at the head of the queue to be successfully sent. In other words it is the length of time a message spends at the head of the queue.

For the case of two networks, the MIN processing delay must be added to the protocol delay. Other than this the two cases are the same, as shown in Figure 4.1. The delay at the server's host, after the message is successfully received, is not considered.

The simulation program is completely event-driven. It begins at time 0 with a **send message** event. When only using one network, this creates an **enter queue** event for network 0 at time + protocol delay. When using more than one network, then the network with the shorter queue is selected. An **enter queue** event is generated for time + protocol delay, using the protocol delay for the chosen network. When an **enter**

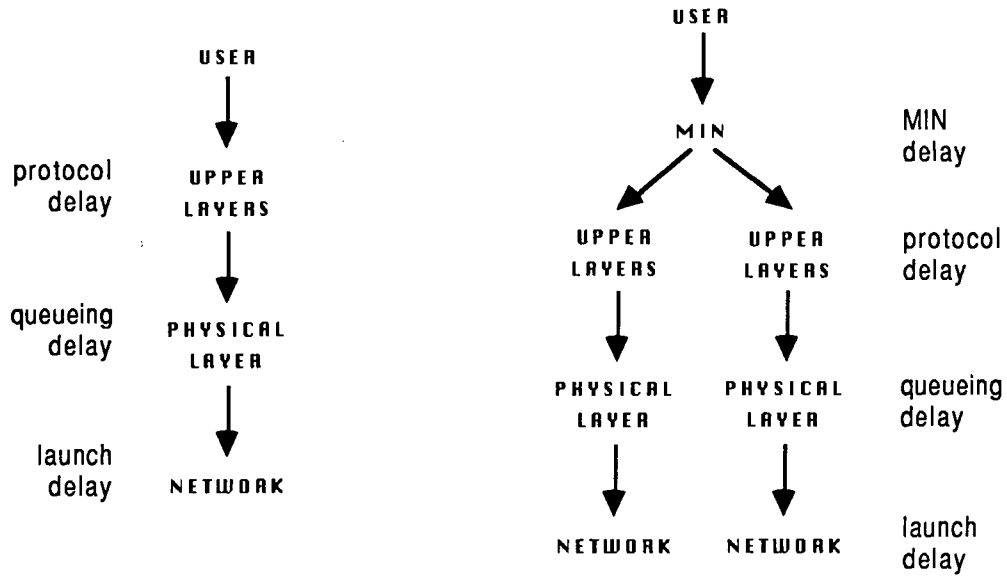


Figure 4.1: Portions of Time Required to Send a Message

queue event occurs, the queue length is incremented. If the queue was previously empty, then a **start launch** event is generated. Also, if the queue is not yet full, then a **send message** event is immediately created, simulating send ahead. A **start launch** event generates a **successful send** event at time + launch delay, using the launch delay for that network. When a **successful launch** takes place, the message at the head of the queue is removed by decrementing the queue length. If this does not empty the queue, then a **start launch** event is created, to start sending the next message in the queue. Also, if the queue was previously full, then a **send message** event is generated, to simulate the un-blocking of the user process. The simulation ends when the specified number of messages have been successfully sent.

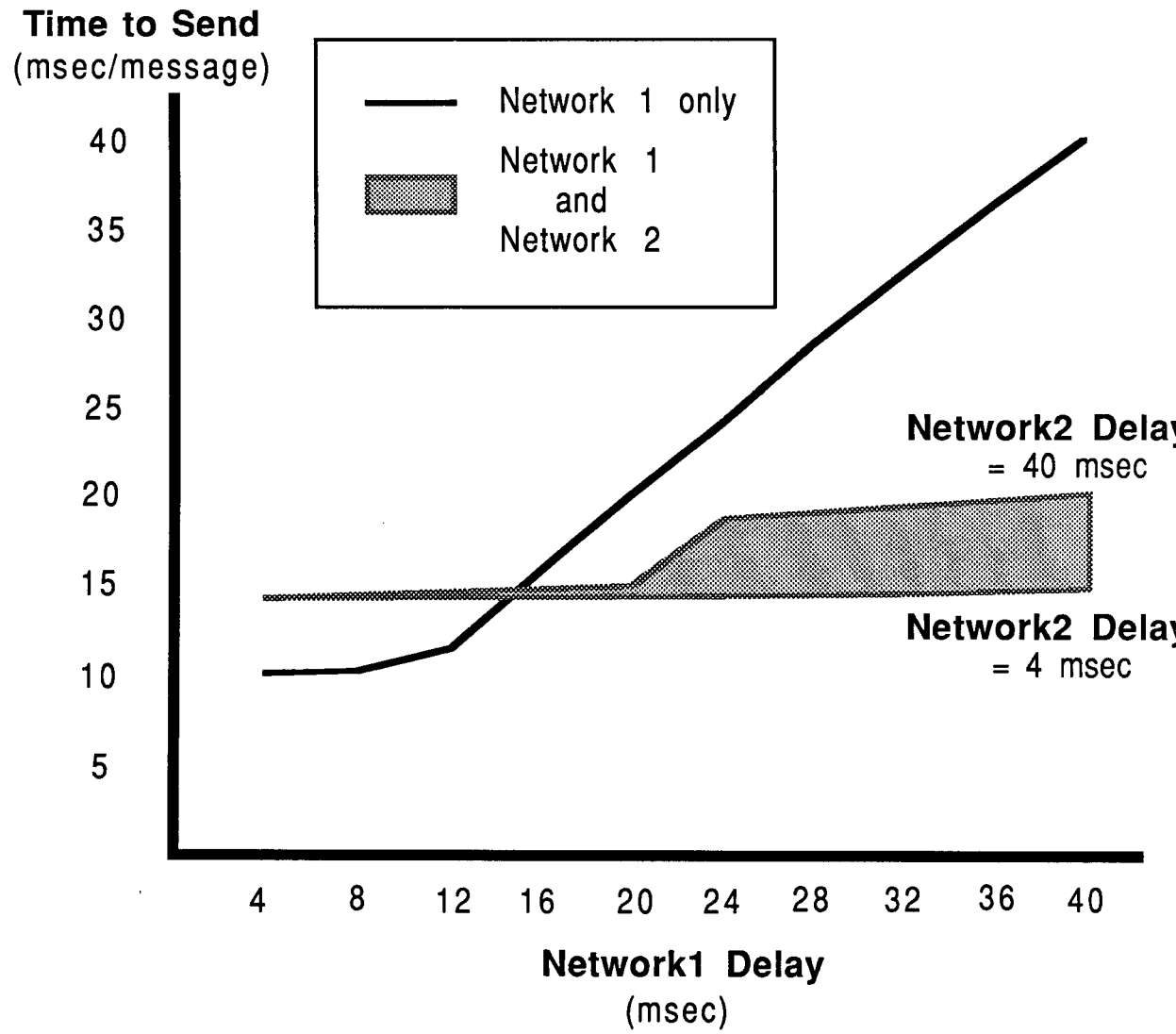


Figure 4.2: Simulation Results For Sending On One Network and Two Networks

The simulation program was executed using various values for the launch delays for each network. However, rather than using a constant launch delay value for each execution, random values uniformly distributed about a mean was used instead, to simulate the variability of network loads and occurrence of transmission errors. A different mean is used for each simulation run. The results are depicted below. The time is measured in units of milliseconds. The MIN delay is taken from the previous measurements as 5 milliseconds while the protocol delay is 10 milliseconds. A total of 100 messages were sent.

From Figure 4.2 one observes that above a certain network delay (network load) the time required to send a message is less using two networks than using a single network. Below this threshold the high MIN overhead exceeds the gain in using two networks. Above this threshold the time to send a message is no longer bound by the processing time, but rather by the high network delay, so that the benefit from using two networks outweighs the MIN overhead. For our results this threshold value for the network delay is about 15 milliseconds. However, this particular value is not of significance outside of our simplified simulation. Additionally, the deterioration in performance with rising network load is less severe for the two network case than for the single network case. This is an intuitive result since the second network provides additional bandwidth and therefore greater possible throughput.

Total Data Transfered (K bytes)	Message Size (bytes)	Total Time Without MIN (secs)	Total Time With MIN (secs)	Difference in Total Time (secs)	Percentage Difference
64	32	9.87	19.88	10.01	101.5
64	64	5.94	10.96	5.02	84.5
64	128	3.69	6.18	2.49	67.4
64	256	2.33	3.58	1.25	53.4
64	512	1.62	2.28	0.66	40.6
64	1024	0.68	1.50	0.82	119.3
64	2048	0.85	1.39	0.55	64.6
128	32	19.83	39.84	20.01	100.9
128	64	11.89	22.17	10.29	86.5
128	128	9.14	12.40	3.26	35.7
128	256	5.06	7.22	2.16	42.7
128	512	3.68	4.53	0.85	23.0
128	1024	1.58	3.02	1.44	91.1
128	2048	1.63	2.64	1.01	61.7
256	32	40.44	79.17	38.73	95.8
256	64	25.12	43.66	18.55	73.8
256	128	14.80	24.84	10.04	67.8
256	256	9.35	14.32	4.97	53.1
256	512	6.51	9.09	2.58	39.6
256	1024	3.25	6.01	2.76	84.8
256	2048	2.95	5.34	2.39	80.8

Table 4.1: Total Transfer Time for One-way Client to Server

Message Size (bytes)	Number of Messages	Time Per Message Without MIN (msecs)	Time Per Message With MIN (msecs)	Difference (msecs)
32	2048	4.82	9.71	4.89
64	1024	5.80	10.70	4.90
128	512	7.21	12.07	4.86
256	256	9.11	13.97	4.86
512	128	12.64	17.77	5.14
1024	64	10.70	23.48	12.77
2048	32	26.48	43.59	17.11
32	4096	4.84	9.73	4.89
64	2048	5.81	10.83	5.02
128	1024	8.93	12.11	3.18
256	512	9.88	14.10	4.22
512	256	14.38	17.69	3.31
1024	128	12.34	23.59	11.25
2048	64	25.47	41.17	15.70
32	8192	4.94	9.66	4.73
64	4096	6.13	10.66	4.53
128	2048	7.23	12.13	4.90
256	1024	9.13	13.99	4.85
512	512	12.71	17.74	5.03
1024	256	12.71	23.48	10.77
2048	128	23.05	41.68	18.63

Table 4.2: Time Per Message for One-way Client to Server

Message Size (bytes)	Total Number of Messages	Time Per Message Without MIN (msecs)	Time Per Message With MIN (msecs)	Difference (msecs)
32	4096	9.79	13.37	3.58
64	2048	10.10	13.99	3.90
128	1024	10.23	15.50	5.27
256	512	13.53	17.26	3.73
512	256	16.95	21.02	4.06
1024	128	17.15	21.76	4.61
2048	64	24.84	40.39	15.55
32	8192	10.20	12.10	1.90
64	4096	8.58	13.04	4.46
128	2048	10.41	14.56	4.16
256	1024	12.51	16.75	4.24
512	512	16.86	20.94	4.08
1024	256	17.33	21.84	4.50
2048	128	24.92	40.76	15.84
32	16384	7.67	12.07	4.41
64	8192	8.65	12.96	4.31
128	4096	10.37	14.53	4.16
256	2048	12.70	16.80	4.09
512	1024	17.12	21.11	3.99
1024	512	17.34	21.85	4.51
2048	256	25.07	40.75	15.68

Table 4.3: Time Per Message for Client to Server with Reply

Chapter 5

Conclusions

This thesis has explored the idea of using multiple local area networks for increasing the reliability and improving the performance of communication in a local environment. It has presented the motivations for researching this topic, and dealt with various issues relevent to the design of a software interface which simplifies the user's task of utilizing multiple networks. Also, the details of an implementation here at the University of British Columbia of this software interface, which we have termed MIN, were discussed. Measurement results on the overhead incurred by using MIN was reported. This chapter draws a few conclusions on what has been done so far and suggests some possible additional work which could be carried out in the future.

The implementation of the MIN software did not require too much effort. With the reasonably reliable and easy to use 4.2 BSD IPC services, only 2 man-months of work were needed to write and debug the code. It would be interesting to try to implement MIN on another system which does not provide a connection-oriented communication

services, to discover how much more difficult the task would be. The decision to provide a message-based rather than a stream I/O type of interface also contributed to the ease of implementation.

It was unfortunate that we were unable to obtain measurements using the Cambridge Ring. If more machines were available so that we could sufficiently load down the networks, then it would be feasible to make measurements of MIN's performance using two networks. Perhaps there is some means for artificially creating heavier loads on the networks. One possibility is to use a reasonably fast machine which is not booted up with a normal operating system, but is instead running some dedicated software which uses up empty ring slots or jams the Ethernet bus to the desired degree.

If the above suggestions cannot be achieved, then an alternative is to improve the simulation program described in Section 4.2. For example, some measurement results can be obtained for the actual protocol delays for the protocols used for the Ethernet and Cambridge Ring. The fraction of the MIN overhead used for CPU cycles and the fraction caused by having to send a header can be determined. These figures can then be applied to the appropriate points in the simulation program's flow of events, rather than merely adding the total MIN delay to the protocol delay at the start of sending the message. Furthermore the simulation program's flow of events may be made more detailed to better reflect what actually takes place in the system.

The header for each message is required because messages sent on different net-

works may arrive out of order at the destination. This is a consequence of having the **min_send** primitive non-blocking in nature. If min_send were made blocking instead, then a sender cannot have 2 messages in flight simultaneously, and the header would no longer be necessary. An acknowledgement from the receiver is required for every message. However, some means must then be devised for distinguishing an incoming message from an acknowledgement. In our implementation environment this problem can be solved by using separate sockets for sending messages and acknowledgements. This is not a general solution, though. Additionally a non-blocking send would mean that a process is unable to send ahead, which will reduce performance by eliminating parallel operations of sending and launching messages.

Another design decision which merits some discussion is that of making a path selection for every message to be sent. Alternatively the MIN software can choose a network at connection time, and send all subsequent messages for this MIN connection on the chosen network. Only when a transmission or reception failure occurs on this network is another network chosen. This would remove the need to check the load of each network every time a message is to be sent. Furthermore, since network loads do not remain constant for long periods of time, the initial network selection should not be based on network loads either. This means that network loads are not used for path selection at all, thus making it unnecessary to keep track of the network loads. However, since we use the length of the outgoing queue as an estimate of the network

load, this does not gain us much. It does not take very much work to maintain and check the queue lengths. Nonetheless, if performance is not an important consideration, then ignoring network loads would be an acceptable means for further simplifying the implementation task. In fact, this is one aspect of MIN which does not have to be the same at all hosts. A MIN user which ignores network loads can still communicate with a MIN user which considers network loads.

We use the length of the outgoing queue at the lowest layer - the physical layer - as an estimate of a network's load, since all messages must eventually pass through this layer before reaching the transmission medium. However, to consider *only* the queue at this layer may not always be correct. Even if a network has a shorter physical layer queue than another network, this network may have many more messages queued up in the higher layers. Since messages queued in any protocol layer contributes delay, the message queues in all the protocol layers should be considered. However, as messages in higher layers may be split into several packets or frames when they are passed to lower layers, it is unknown as to how to weigh the queue length contributions from the various layers. This is similar to the problem of comparing queue lengths in different physical layer protocols for different networks, mentioned in Section 2.8.

This thesis has examined in some detail a number of design issues relevent to using multiple LANs. Some but not all of the discovered problems have been adequately solved. More problems arose than was anticipated at the beginning of the research

effort, but this is often the case with relatively unexplored topics. Our implemented software has been found to incur fairly high overhead. Nonetheless a simulation study, albeit greatly simplified, suggests that under heavy network loads this overhead is outweighed by improved performance. In any case, reliability is certainly enhanced by having more than one available network. At this time, though, it is still unclear as to whether MIN is a concept of practical value which may someday be put to general use. Further investigation is in order.

Bibliography

[Boei85] Boeing, *Technical and Office Protocols*, Specification Version 1.0, Nov. 1985

[Cabr85] L.F. Cabrera, M.J. Karels, D. Mosher, *The Impact of Buffer Management on Networking Software Performance in Berkeley Unix 4.2BSD : A Case Study*, U.C. Berkeley, report #UCB/CSD 85/247, Jun. 1985

[Chan83] S.T. Chanson, A. Kumar, A.V. Nadkarni, *Performance of some Local Area Network Technologies*, Proc. of Spring Compcon Conference, San Francisco, Mar. 1983

[Chan84] S.T. Chanson, K. Ravindran, S. Atkins, *An Efficient Transport Protocol for Local Area Networks*, Technical Report, U. of British Columbia, Dec. 1984

[Chan85] L. Chan, *Implementation of the Cambridge Ring Protocols on the Sun Workstation*, M.Sc. Thesis, U. of British Columbia, Jul. 1985

[Chan86] S.T. Chanson, K. Ravindran, *Host Identification in Reliable Distributed Kernels*, Technical Report 86-5 (draft), U. of British Columbia, Apr. 1986

[Cher84] D.R. Cheriton, *The V Kernel : A Software Base for Distributed Systems*, IEEE Software, Vol. 1, #2, pp. 19-42, Apr. 1984

[Dall81] I.N. Dallas, *Transport Service Byte Stream Protocol*, U. of Kent at Canterbury Computing Lab. report #1, Aug. 1981

[DARP81a] DARPA Internet Program Protocol Specifications, *Transmission Control Protocol*, RFC 793, Information Sciences Institute, USC, CA, Sep. 1981

[DARP81b] DARPA Internet Program Protocol Specifications, *Internet Protocol*, RFC 791, Information Sciences Institute, USC, CA, Sep. 1981

[GM85] General Motors, *MAP Specification 2.1*, GM Technical Centre, Warren, Michigan, Mar. 85

[ISO81] Int'l Standards Organization Draft Standard 7498, *Open Systems Interconnection Basic Reference Model*, ISO/TC97/SC16, 1981

[Leff83] S.J. Leffler, R.S. Fabry, W.N. Joy, *4.2 BSD Interprocess Communication Primer*, Computer Systems Research Group, U.C. Berkeley, Mar. 83

[Limb84] J.O. Limb, *Performance of Local Area Networks at High Speed*, IEEE Communications Mag., Vol. 22, #8, Aug. 1984, pp. 41-45

[Metc76] R.M. Metcalfe, D.R. Boggs, *Ethernet : Distributed Packet Switching for Local Computer Networks*, Communications of the ACM, Vol. 19, #7, July 1976, pp. 395-404

[Neil85] W.J. Neilson, U.M. Maydell, *A Survey of Current Local Area Network Technology and Performance*, INFOR, Vol. 23, #3, Aug. 1985, pp. 215-247

[Tane81] A.S. Tanenbaum, *Computer Networks*, Prentice-Hall Inc., 1981

[3COM82] 3COM Corporation, *Multibuf Ethernet Controller Reference Manual*, May 82

[Toltec] Toltec Computer Limited, *Toltec's Cambridge DataRing*

[Tsao84] C.D. Tsao, *Defining LAN Environments and User Needs*, IEEE Communications Mag., Vol. 22, #8, Aug. 1984, pp. 7-11

[Vuon83] S.T. Vuong, S.T. Chanson, R.K.Y. Lee, *A Hybrid Local Area Network for Efficient and Reliable Communication*, Proc. of Int'l Electronics and Electrical Conference and Exposition, Toronto, 1983

[Wilk79] M.V. Wilkes, D.J. Wheeler, *The Cambridge Digital Communication Ring*, Proc. of Local Area Communication Network Symposium, Mitre Corp., May 1979, pp. 46-61

[Zwae85] W. Zwaenepoel, *Protocols for Large Data Transfers over Local Networks*, Proc. of 9th Data Communications Symposium, Whister Mtn., B.C., Sep. 1985, pp. 22-32