THE DESIGN OF A PACKET-SWITCHED LOCAL AREA NETWORK

by

Richard C.S. Morling

A thesis in partial fulfilment of the requirements for the Degree of Doctor of Philosophy

THE CITY UNIVERSITY

Centre for Information Engineering

October 1988

DEDICATION

This thesis is dedicated to the memory of my [redacted]

CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Prof. A.C. Davies, for his patient encouragement, without which this thesis would not have been completed, and his helpful and constuctive suggestions. I would also like to thank [redacted] for his support and direction. I owe a debt of gratitude to all those who have worked on Project MININET. In particular, my thanks goes to [redacted] for their helpful suggestions. Finally, I should like to thank [redacted] for binding up the split infinitives and separating the "or's" from the "nor's".

ABSTRACT

MININET is a local area network designed for instrumentation and other real-time applications. It is a true store-and-forward network using small fixed-length packets. A very high degree of transparency is required, such that the Network Service can be totally invisible to the user devices. The design of the network is based around a hierarchical architectural model which is similar to, but not identical with, the ISO OSI Reference Model. Its small 32-bit packet size was a logical consequence of the transparency and real-time service requirements. This size was found to be optimum for average user message lengths of around 13 bits, as far as the buffer utilization efficiency was concerned. A simple, but robust, full-duplex data link protocol, which avoids sequence errors and uses only a single sequence number field to interlock the packet stream, was developed. Network congestion is avoided by means of a flow control algorithm, which uses active backpressure vectors and a separate buffer allocation for each destination node, to guarantee freedom from store-and-forward deadlock. A highly reliable, half-duplex, end-to-end protocol providing a Transport Service for the network management entities, has been developed. The network compatible DIM intermediate interface has been specified, together with DIM-CPC, its basic user protocol providing flow control, initialization and error recovery procedures. A routing protocol, that maintains packet sequency even in the event of node or link failure, has been developed. This distributed algorithm constructs a separate tree rooted at each destination node in the network. It uses short messages transferred only between adjacent nodes. A quad-phasic update cycle is used to guarantee loop freedom at all times and to flush old pathways before routing changes are made. Thus, packet sequency is maintained without any packets being dropped. A high-speed implementation of the network Station has been designed and constructed. This uses a two-dimensional polling technique in order to maintain fairness, whilst ensuring that no blockages occur within the node. A design technique has been pioneered for the construction of PROM-based system controllers which are exceptionally agile.

Chapter 1

INTRODUCTION

This thesis describes the development of ***MININET***, a local area network (LAN). The requirements, design decisions and implementation of the network are described. It is the aim of this thesis, not just to give an account of this work, but also to highlight the lessons learnt in the design process and to describe those concepts, protocols and techniques which are of general applicability.

A brief overview of the network, its requirements, its relationship with other networks and a short history of the project are given in this chapter. In Chapter 2, the various network services and protocols are explained in the context of an overall hierarchical model of the network. The ***DIM intermediate interface***, used to access the network, and the routing management algorithm are defined in Chapters 3 and 4 respectively. An implementation of the network Station is described in Chapter 5. In Chapter 6, the design decisions are critically reviewed and the lessons learnt from the project are discussed.

1.1 OVERVIEW AND TERMINOLOGY

MININET was conceived as a LAN intended for high-speed instrumentation applications [MORL75]. These include laboratory automation, process control, high energy physics and other areas generating high-speed, real-time data. In addition, the properties of the network make computer-computer communication particularly easy to implement.

The key objective underlying the development of the network is that computers and peripherals should be able to communicate through the network just as though they are connected directly together. User devices are connected to the network through ***ports*** which are physical interfaces into the network. Originally, the network was designed using a single port interface standard, DIM, which is described in Chapter 3. This is a one-to-one interface designed for high-speed instrumentation

and network applications. It includes 16 bidirectional data lines, with an additional qualifier line, the ***data class flag***, which distinguishes between ***data class*** and ***control class*** transfers. The latter class is used for the exchange of status and command information. The network service was originally defined in terms of this interface and the service boundary was therefore physically located at the DIM port socket. Other possible types of port interface are an IEEE-488 (IEC 625) instrumentation bus interface [IEC 79], which enables the bus to be "stretched" across the network, and a speech port, which provides voice communications (Section 2.5.2). To cater for these heterogeneous interfaces, the network service is defined in terms of communication between ports.

In order to provide this high degree of transparency, the Network Service provides a ***Virtual Connection*** between ports, which means that a data word entering a port at one end of the connection will emerge from the port at the other and vice versa. This is done by using a very small packet of only 32 bits which contains a single 16-bit word of user information as well as the Data Class Flag. Thus, each word is despatched immediately it enters the network and presented to the user immediately it reaches the other end of the Virtual Connection. Consequently, MININET has been aptly dubbed a **word** switching network.

The network can accommodate up to 64 nodes interconnected in an arbitrary topology by ***channels*** as shown in Figure 1.1. These channels may be point-to-point half-duplex or full-duplex data links or ***multi-node channels*** such as rings or buses. Implementations of the point-to-point channels have used the ***MININET link protocol (MLP)*** in order to provide a sequential Data Link Service. MININET is a true network, in the ISO Open System Interconnection (OSI) Reference Model sense [ISO 82], because communicating nodes do not have to be connected to a common data link, but may transfer information via one or more intermediate nodes. There are two basic types of node in the network. One, the ***Station***, provides user access into the network via the network ports. The other, the ***Exchange***, provides, in addition, a store-and-forward relay function. Nodes connected to the same channel are said to be ***adjacent***.

The state information associated with each Virtual Connection is held by the nodes at each end of the connection. Transportation of packets, between these end-point nodes, is provided by the ***Packet***

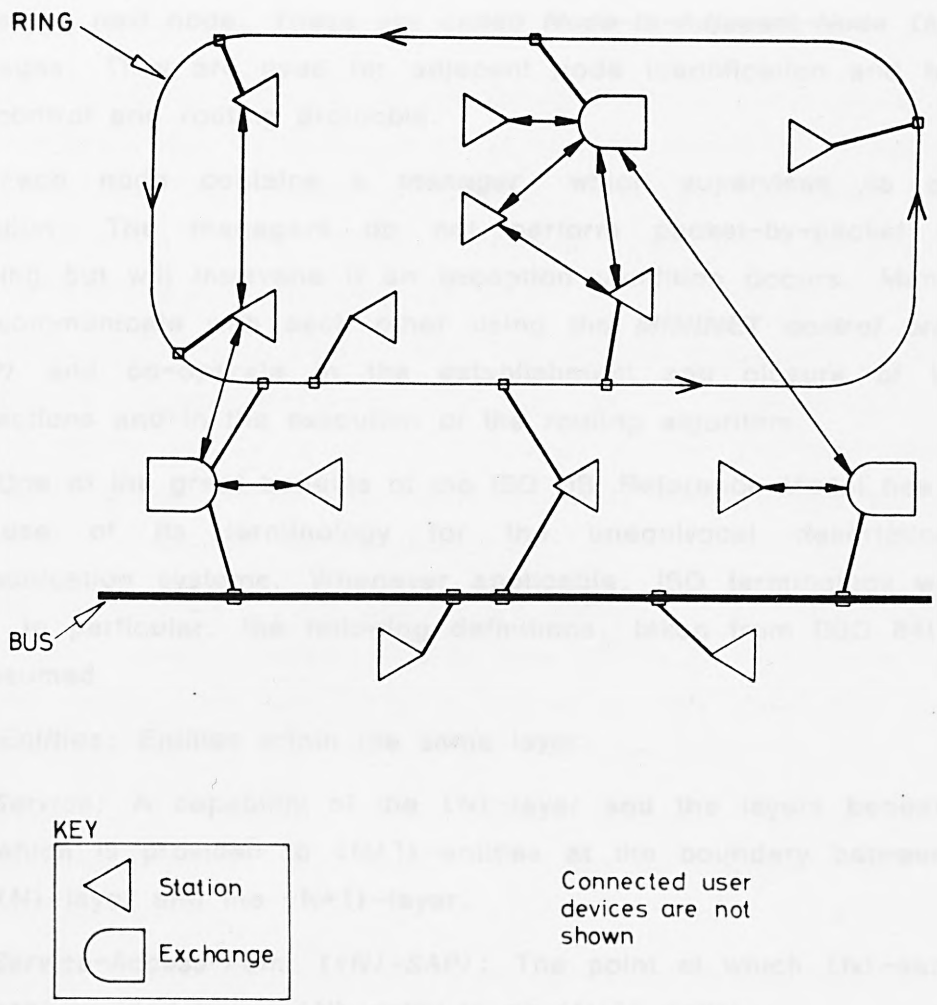


Figure 1.1: An Example MININET

Delivery Service which maintains a ***path*** between each and every node. Virtual Connections are multiplexed along these paths. Underlying the Packet Delivery Service are the flow control and routing management protocols. Network congestion is avoided by maintaining, in each Exchange, separate buffer allocations for each destination node. Flow control in and out of these buffers is effected by means of ***Back Pressure Flow Vectors (BPVs)*** transmitted to all adjacent nodes. Each element of the vector acts as a stop-go binary semaphore pertaining to a single destination node. For each node in the network the routing management algorithm, described in Chapter 4, constructs a tree rooted at the destination node. The algorithm guarantees to remain loop free at all times and to minimize the channel weighted distance from each node to the root.

In addition to transporting packets sequentially between adjacent

nodes, the channels also carry special messages which only travel as far as the next node. These are called ***Node-to-Adjacent-Node (NTAN)*** messages. They are used for adjacent node identification and by the flow control and routing protocols.

Each node contains a ***manager***, which supervises its overall operation. The managers do not perform packet-by-packet traffic handling but will intervene if an exception condition occurs. Managers can communicate with each other using the ***MININET control protocol (MCP)*** and co-operate in the establishment and closure of Virtual Connections and in the execution of the routing algorithm.

One of the great benefits of the ISO OSI Reference Model has been the use of its terminology for the unequivocal description of communication systems. Whenever applicable, ISO terminology will be used. In particular, the following definitions, taken from [ISO 84], will be assumed.

Peer Entities: Entities within the same layer.

(N)-Service: A capability of the (N)-layer and the layers beneath it, which is provided to (N+1)-entities at the boundary between the (N)-layer and the (N+1)-layer.

(N)-Service-Access-Point ((N)-SAP): The point at which (N)-services are provided by an (N)-entity to an (N+1)-entity.

(N)-Protocol: A set of rules and formats (semantic and syntactic) which determine the communication behaviour of (N)-entities in the performance of (N)-functions.

(N)-Relay: An (N)-function by means of which an (N)-entity forwards data received from one correspondent (N)-entity to another correspondent (N)-entity.

(N)-Protocol-Control-Information: Information exchanged between (N)-entities to co-ordinate their joint operation.

(N)-User-Data: The data transferred between (N)-entities on behalf of the (N+1)-entities for which the (N)-entities are providing services.

(N)-Protocol-Data-Unit ((N)-PDU): A unit of data specified in an (N)-protocol and consisting of (N)-protocol-control-information and possibly (N)-user-data.

(N)-Interface-Control-Information: Information transferred between an (N+1)-entity and an (N)-entity to co-ordinate their joint operation.

(N)-Interface-Data: Information transferred from an (N+1) entity to an (N)-entity for transmission to a correspondent (N+1)-entity, or conversely, information transferred from an (N)-entity to an (N+1)-entity after being received from a correspondent (N+1)-entity.

(N)-Interface-Data-Unit ((N)-IDU): The unit of information transferred across the service-access-point between an (N+1)-entity and an (N)-entity in a single interaction. Each (N)-IDU contains (N)-interface-control-information and may also contain the whole or part of an (N)-service-data-unit.

(N)-Service-Data-Unit ((N)-SDU): An amount of (N)-interface data whose identity is preserved from one end of an (N)-connection to the other. Data may be held within an (N)-connection until a complete (N)-SDU is put into the (N)-connection.

Multiplexing: A function within the (N)-layer by which one (N-1)-connection is used to support more than one (N)-connection.

Segmenting: A function performed by an (N)-entity to map one (N)-SDU into multiple (N)-PDUs.

Reassembling: The reverse function to segmenting.

Blocking: A function performed by an (N)-entity to map multiple (N)-SDUs into one (N)-PDU.

Concatenation: A function performed by an (N)-entity to map multiple (N)-PDUs into one (N-1)-SDU.

The ISO definition of a connection does not quite correspond to MININET's Virtual Connection. These differences are discussed in Section 2.1.3.

This thesis is primarily concerned with the 4 lowest layers of the Reference Model. These are:

Layer 1: The *Physical Layer* provides the transmission encoding/decoding, signal conditioning and timing generation/recovery functions necessary to convey information bits between Layer 2 entities through the physical medium for

interconnection (e.g. optical fibre or coaxial cable). The SDU of the Physical Layer is usually 1 bit in length.

Layer 2: The *Data Link Layer* enhances the Physical Layer Service with block synchronization, flow control and error detection and recovery procedures to enable adjacent Layer 3 entities to exchange PDUs in an orderly manner.

Layer 3: The *Network Layer* provides relaying and routing functions which enable Transport Layer entities (located in the end systems) to communicate transparently. The choice and use of Data Link resources are made invisible to the Transport entities. The Network Layer can be sublayered using the principle of recursive extension [ECMA82] to enable *Subnetworks* to be interconnected to form larger *Global Networks*. In this case, the users of the Subnetwork Service are not the Transport Layer entities but the Global Network entities.

Layer 4: The *Transport Layer* provides, where necessary, the end-to-end protocols such as multiplexing, flow control and error recovery to enhance the Network Service to the level required by the end user.

1.2 AIMS AND REQUIREMENTS

Instrumentation applications present a different and somewhat more onerous set of requirements on the Network Service compared with computer-to-computer and office automation environments. The major requirements may be summarized as:

- heterogenous device handling capability;
- ultra-high transparency;
- real-time operation;
- reliability;
- reconfigurability.

The reasons for these requirements and their effect on the network design choices are discussed in the following sections.

1.2.1 Heterogeneity

There should be no constraint on the type of device connected to the network. In particular, it must be able to service not only "intelligent" devices such as mini- or microcomputers but also "ignorant"

devices such as analog-to-digital converters (ADCs), digital-to-analog converters (DACs) or terminals. In this context, an intelligent device is one which can be made aware of the network and can be adapted to handle the procedures required to access it. On the other hand, an ignorant device cannot be modified to comply with the demands of a user-invasive network. It should be noted that the mere presence of a microprocessor embedded in an instrument does not necessarily mean that it is intelligent in this sense, unless the microprocessor is, or can be, programmed to handle a network access protocol.

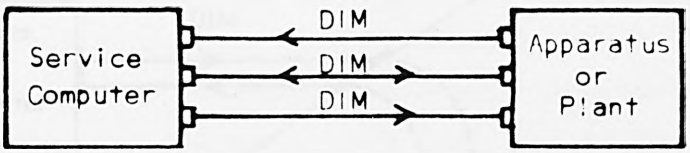
Secondly, there should be no dependence on any particular device manufacturer. User devices from different manufacturers may be freely connected to the network. Furthermore, the network should be able to support the simultaneous communication between different pairs of user devices, with each pair using its own private protocol without interference.

Finally, the network should simultaneously support a population of user devices communicating at widely different rates without lockout or network hogging. Examples range from low speed terminal and process control traffic, operating at information rates below 10k bits/sec, through computer-computer communications to high speed laboratory instrumentation operating in excess of 1M bits/sec. (Note that effective user throughputs are being discussed, not intra-network line rates.) Broadly speaking, user traffic may be divided into two distinct classes. Many devices operate in ***handshake mode*** where a device waits after transmitting a short message until it has received an acknowledging reply from the destination before transmitting the next message. For this class, the effective throughput is controlled by the end-to-end transport delay of the network. Other devices operate in ***burst mode*** where a relatively long message is transmitted through the network with no end-to-end acknowledgement until the end of the burst. For this class, the effective user throughput is dominated by the throughput of the network.

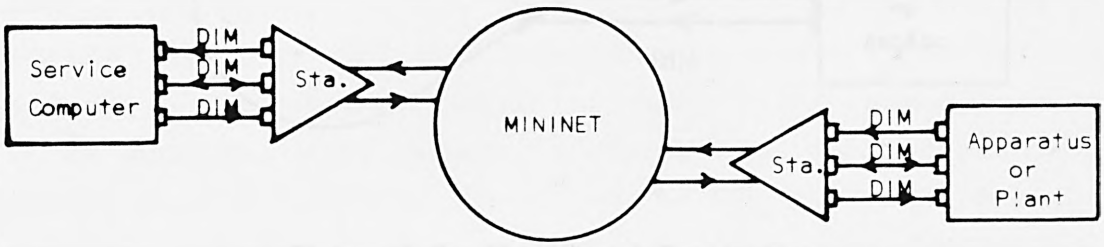
1.2.2 Transparency

As far as the user is concerned, there should be no operational difference between the connection of two devices directly together, as shown in Figure 1.2a, or the connection of two devices through the

network, as shown in Figure 1.2b. This degree of transparency applies



(a) Directly Connected Equipment



(b) Equipment Remotely Connected via the Network

Figure 1.2: Direct and Indirect Connection of User Devices

from high level software procedures right down to the hardware plug and socket level. Effectively, the network is to emulate a length of cable! To provide this service, the network must establish a Virtual Connection between two ports so that information transferred into one port emerges from the other end and vice versa. Thus, the Virtual Connection acts as a pair of *pipes* conveying information sequentially in either direction (Figure 1.3). The exact nature of this Virtual Connection and its relation to connections as defined in the ISO OSI Reference Model are discussed in Section 2.5.1 and Section 2.1.3.

Since the user protocol is unknown to the network in the same way as it would be unknown to a cable, the network has no way of delimiting user messages other than to the IDU. In addition, the delay associated with blocking these data units into a larger packet is undesirable in a real-time network. For these reasons, the network should despatch data towards the destination as soon as it is passed across the port interface. This implies that the Network-PDU should only contain one

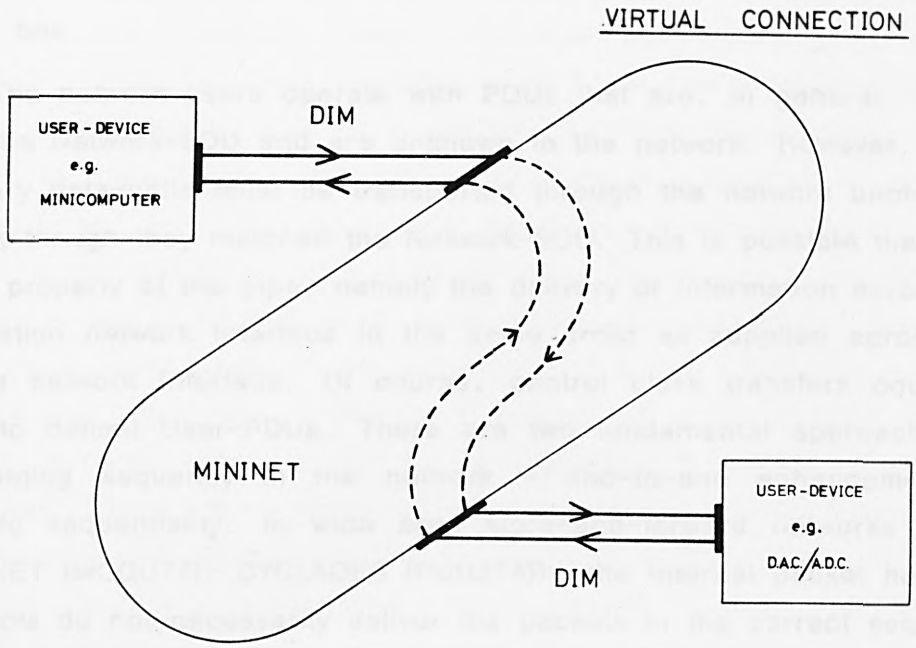


Figure 1.3: The Virtual Connection

Network-SDU and that the Network-SDU is equal to the largest Network-IDU likely to be used.

The maximum size of the IDU should be such that the "basic unit" of data handled by the majority of instrumentation devices should be transferred in a single operation. For example, the "basic unit" of data for a terminal is one character, for an ADC it is one sample and so on. This requirement implies that the size should be greater than 8 bits as ADCs and DACs frequently have a sample size of 10, 12 or even 16 bits. A natural and convenient choice is therefore 16 bits, as it is large enough to accept the output of most data acquisition equipment, while a larger data unit (say 24 or 32 bits) would tend to be mostly unused and so not cost effective. The relationship between message size, packet size and efficiency is discussed in Section 2.2.2.

User devices usually need to exchange control information such as peripheral status or computer commands in addition to the transfer of "data" information. The network should be able to transfer this control information without making any code restrictions on the data transfers which would result in an effective loss of transparency. The simplest means of providing this is to add a flag bit to the Network-SDU. This bit

can be used to distinguish between data class and control class transfers. Thus, the overall size of the network-SDU should effectively be 17 bits.

The network users operate with PDUs that are, in general, bigger than the Network-SDU and are unknown to the network. However, these arbitrary data-units must be transported through the network unchanged just as though they matched the Network-SDU. This is possible thanks to a key property of the pipe, namely the delivery of information across the destination network interface in the same order as supplied across the source network interface. Of course, control class transfers could be used to delimit User-PDUs. There are two fundamental approaches to maintaining sequency in the network - end-to-end enhancement or intrinsic sequentiality. In wide area store-and-forward networks (e.g. ARPANET [MCQU77], CYCLADES [POUZ74]), the internal packet handling protocols do not necessarily deliver the packets in the correct sequence to the destination node. Instead, the node must resequence arriving packets prior to delivering the packets to the user. This message reassembly time is undesirable in a real-time environment. Furthermore, a sequence number must be carried in every packet header. Since the Network-PDU is small, this results in an unacceptably large packet overhead. The other approach maintains packet sequency throughout the network so that reordering is unnecessary. This has an impact on two areas of protocol design. Obviously, one is the routing protocol, where dynamic path changes could lead to sequence errors. Somewhat less obviously, it also affects the data link protocol, as retransmission of a previously corrupted packet could also lead to a sequence error. A routing algorithm specifically developed to maintain intrinsic packet sequency is described in Chapter 4 and a link protocol which avoids sequence errors during transmission is described in Section 2.3.2.

1.2.3 Real-Time Operation

Computers (and human terminal users, for that matter) are relatively easily satisfied, as far as the response time and throughput of the network are concerned, compared with the real-time environment of instrumentation systems where the desired response time is measured in microseconds rather than milliseconds. As already discussed, the propagation delay across the network is frequently much more important than network throughput. The properties of the pipe mean that it can be

modelled as a finite-length, *first-in, first-out (FIFO)* queue combined with a time delay. The actual value of the time delay and the queue throughput are quantities which vary with the network traffic and connectivity. For real-time applications, it is necessary to place bounds on these variations. There are a number of ways to specify these limits. If the user cannot buffer more than one word at a time (e.g. a single ADC sample), then the worst case maximum propagation delay of the Virtual Connection is the critical parameter if the device is operating in handshake mode, or the worst case maximum transmission interval if operating in burst mode. If the real-time device has a buffering capacity of N words, then the constraint is relaxed to a worst case moving window N-block average of the propagation delay or throughput respectively. Note that buffering within the network ports will average out any intra-network fluctuations in throughput and so only the block averaged throughput of the network is significant. However, nothing can be done inside the network interface to mask any fluctuations in propagation delay. Indeed, in order to reduce propagation delay, queuing should be avoided wherever possible [MCQU77].

As discussed in Section 1.2.1, the actual throughput requirements of real-time devices vary enormously, but a useful benchmark for the fastest likely requirement has been the case of high-fidelity stereo audio analog/digital conversion. This generates two 16-bit samples at a 50kHz rate which corresponds to a throughput requirement of 100k packets per second. This has been adopted as the target throughput figure for the network. It is reasonable to expect to maintain such a sustained rate only in burst mode. For handshake mode, the effective rate depends very much on the number of hops in the path. A delay of 25μs per hop is not unreasonable, which for a single hop would correspond to a user throughput of 20k words per second. These figures depend, of course, on the channel throughput and assume that a high-speed channel, such as a coaxial cable based system, is being used. The presence of a modem or low cost optical fibre system in the path would necessarily reduce the effective throughput. Nevertheless, the designer of the Stations and Exchanges must assume that their channels could each be operating up to 100k packets per second.

An implicit assumption of the discussion, so far, is that the network has been only lightly loaded. The converse of this situation is when a part, or all, of the network becomes saturated by many devices

attempting to communicate at the same time. An important requirement of the network, when handling this congestion, is the ***fairness criterion.*** This demands that, whenever a network resource – such as a channel, buffer space or processor time – is in contention, the resource should be shared equally amongst those requiring service. In particular, the network must not allow burst mode transfers to swamp handshake mode traffic.

1.2.4 Reliability

A key method of providing fault tolerance is the provision of redundancy. It should be possible to configure the network so that it can recover automatically from failure of nodes or links. This implies that the topology of the network must allow alternative pathways and that essential network management functions should not be centralized.

A ring or bus topology does not meet this requirement without duplication. In fact, from a reliability point of view, a ring or bus may be considered to be a star system, with the ring or bus, together with their couplers, forming the centre of the star. Hence, a failure of any of these central elements causes total communication system failure. On the other hand, a network, which could be arbitrarily configured into any topology, can be designed so that, even if a section of the network becomes isolated, each part can operate independently. The network should maintain communication pathways between all nodes regardless of the node or channel failures, providing, of course, that it is still physically possible. Furthermore, recovered nodes and channels should be reincorporated into the network automatically and a partitioned network should recombine smoothly when they are physically re-joined.

The end nodes and ports of a Virtual Connection form unavoidable single points of failure as far as that connection is concerned. Complete redundancy can only be provided by the user setting up more than one Virtual Connection between different end nodes, as well as providing duplicate user equipment. For example, a completely redundant remote temperature measurement would require not only duplicate network paths, end Stations and ports, but also duplicate temperature transducers.

The various types of packet error can be classified in order of importance to the user, on a scale ranging from outright disaster to

minor irritation, as follows:

1st. message delivered to wrong port;
2nd. packet sequence error;
2nd. corrupted message;
3rd. duplicated message;
4th. unreported lost packet;
5th. reported lost packet.

These form the relative priorities of the error procedures of the network. The misdelivery of a packet, to the incorrect destination, is potentially very serious because it could, for example, trigger a quiescent device at quite the wrong time, as well as losing the message as far as the correct destination is concerned. Sequence errors and message corruption are close joint second in priority. Sequence errors are included at this level, because many user Transport Layer error detection procedures depend upon a simple checksum or longitudinal parity check, which are insensitive to sequence errors. A corrupted message is equally dangerous, particularly if its user has no end-to-end error checking or where the damaged message appears as a control word. The choice, of whether to give duplication higher priority than loss, is less clear-cut. From a pragmatic point of view, loss of packets may well happen if a node or channel fails completely. Message loss is relatively easily detected by the user using a simple timer based protocol, situated either at the source or at the destination. The network should deliberately drop a packet, if there is any danger of it being duplicated. Furthermore, there are a number of rare situations when the network may deliberately drop packets. These are where there is a danger of sequence errors (Section 4.3.4) or where packet corruption is suspected. If possible, the network should report packet loss to the user.

Channels operating with only marginal signal quality and nodes with intermittent faults can be extremely disruptive to the network as a whole. In addition, there is a greatly increased risk of packet corruption for packets passing through such a node or channel. Therefore, the network should practise *error hardening* by removing, from service, elements of the network that have become marginally operational.

To enhance the overall reliability and to aid fault diagnosis, the performance of the nodes and channels should be monitored continually

by the network management entities. Recoverable errors should be logged by each node, so that suspect modules within the node or the channel can be identified.

Hop-by-hop error detection and recovery should be provided within each data link. The properties and complexity of this protocol can be tailored to the intrinsic error characteristics of the physical medium used, in order to make the probability of an undetected error negligibly small. Quite what error rate is considered negligible is, of course, somewhat arbitrary. A useful yardstick can be found in the requirement for PROWAY [IEC 81], which specified an average undetected error rate of better than one every thousand years of operation! Operating at 100k packets per second, this corresponds approximately to a probability of undetected packet corruption of 3×10^{-16}. Needless to say, error protection, to this degree, cannot be measured by any practical means. However, schemes that achieve this protection in theory, at least make the probability of an undetected error, within the channel itself, negigible. Note, that the probability of framing errors must also be taken into account in arriving at the probability of errors in the channel.

Given the high reliability of the channels, the probability of error, within the channel controller and the rest of the node, cannot be ignored, although it is extremely difficult to quantify. The design of the nodes should be such that a single bit error cannot cause packet misdelivery, packet corruption or sequence errors. This implies that all address and data fields within the node should contain at least one parity check digit.

1.2.5 Reconfigurability

When power is applied to a node, it should automatically configure itself and integrate into the network. The network should adapt dynamically, as nodes and channels become operational. If two isolated parts of the network, which were initialized separately, are eventually joined they should integrate to form a single network automatically. The topology of the network should be able to take any form. When new nodes or channels are added to an existing network, no modification should be necessary to the existing nodes. The nodes themselves should be of a modular construction enabling new ports and channel controllers to be added without difficulty. Only reinitialization of the node, which

occurs on power-up, is required to incorporate the new ports or channel controllers into the node.

As well as the user device characteristics and requirements being highly diverse, the network must be capable of operating over a variety of physical media such as coaxial cable, optical fibres, modem links, etc. In addition, multi-node channels, such as rings or buses, should be accommodated by the network protocols. The diversity of interconnecting media implies potentially large variations in maximum throughput between channels connected to a network node. Slow channels should not retard the operation of the node or other high-speed channels.

The net effect of these requirements is that neither network topology nor number or type of channels or ports attached to the node can be permanently stored in the node manager. Instead, the manager must dynamically obtain this information after initialization and configure itself accordingly.

1.3 RELATIONSHIP WITH OTHER NETWORKS

In recent years, there has been a proliferation of, so called, local area networks. However, most of these are not true networks, but multi-point data links providing a Data-Link Layer Service. They can be divided into two classes. In one type, the physical medium is common to all stations. This usually takes the form of a bus, although passive stars, that have been proposed for use in fibre-optic systems [RAWS78], also fall into this category. The other major type utilizes a Physical Layer relay function in each station. In this case, the usual topology is that of a ring or loop.

The primary application of many recent ring and bus developments has been the expanding field of office automation. In this environment, the network is primarily concerned with the transfer of files, electronic mail and data-base queries and updates. Therefore, the real-time delay requirements are less stringent than in instrumentation applications, being limited to that of human interactions. In addition, the office traffic characteristics are well suited to the use of a connectionless type of service (otherwise known as a Unit Data Service). This is in contrast to the instrumentation situation, where an indefinite (usually large) number of data units have to be transferred during the lifetime of a Virtual

Connection.

The major differences, between the various ring and bus designs, lie in their *access methods* which primarily determine the degree of fairness and flow control exhibited by the designs. It is of interest to examine the various access methods, particularly with a view to their suitability for use as a multi-node channel within MININET.

1.3.1 Loop Access Methods

There have been several attempts to classifiy loop access methods including one by Heger [HEGE78] and another by Penney and Baghdadi [PENN78]. In this thesis, the ring control mechanisms have been divided into four major classes - *token-passing rings*, *slotted rings*, *register-insertion rings* and *centralised ring control*. These are discussed separately.

(1) Token Passing Rings

With this access method, a single *token* circulates around the ring. When a station receives the token it can transmit a message (usually of variable length), prior to passing the token on to the next station in the loop. If the station has no message to transmit, the token is passed on immediately. This technique was first developed by Farmer and Newhall in 1969 [FARM69], after whom it is frequently named. It is also the basis of the access method for the ring version of the IEEE LAN standard 802.5 [IEEE85]. The method is intrinsically fair, with the guaranteed maximum access time proportional to the maximum size of the PDU and the number of stations in the loop. Under light load conditions, the average latency time is equal to half the loop transit period. This loop is, therefore, ideally suited to situations where long messages (relative to the loop transit delay) are to be transported. This is not particularly well matched to the MININET situation which has short, fixed-length packets. Nevertheless, such a ring is quite usable, within MININET, especially where the size of the loop is small. The efficiency (in terms of channel capacity) would be strongly dependent on the header, token and delimiter size.

In the original Farmer and Newhall loop, the management of the loop was centralized with a supervisor station providing bit timing, token initialization and loop closure. However, more recent implementations,

such as proNET [SALW83], have distributed these functions or provide for standby supervisors, as in the case of IEEE Standard 802.5 [IEEE85], removing the need for a supervisor node with the consequent risk of total loop failure. ProNET is of special interest because the absence of a master oscillator in a synchronous ring. Instead, each station contains a crystal stabilized oscillator phase-locked to the received data. Together, the oscillators form a ring of phase-locked loops.

(2) Slotted Rings

In a slotted loop, an integer number of fixed-length (usually short) frames or slots permanently circulate around the ring. Associated with each slot is a full/empty flag. If a station on the ring wishes to send a message, it waits until an empty slot is detected. It then marks the slot full and places the message into the slot. This type of loop was first described by Pierce [PIER72]. In his design, the slot is emptied by the destination station. Such a scheme has two disadvantages. Firstly, each station must buffer the incoming slot sufficiently to read the destination address and full/empty flag prior to onward transmission. In a local area environment, this greatly increases the effective length of the loop. Secondly, it is possible for one station to fill slots continuously, so denying access to all stations between it and the destination. Thus, the method is intrinsically unfair with an indefinite maximum access time. Kropfl's implementation of the Pierce Loop [KROP72] overcame the latter problem, using a special hog prevention control field in the slot headers. However, a more complete solution to these problems was achieved in the Cambridge Ring [HOPP77], by allowing the full slot to complete a complete revolution of the ring, before being set empty by the original transmitter. The latter must release the slot, even if it has another message to transmit. Thus, the station following an active station is guaranteed access after one revolution. In the case of a single slot loop, this method is almost identical to that of a token passing ring. However, with a larger loop, where the loop delay is long compared with the slot length, the multiple slots circulating around the loop are equivalent to a number of tokens. This reduces the loop access time compared with a single token ring. Therefore, slotted rings are particularly well suited to the short, fixed-length messages of MININET. The fairness and guaranteed maximum access time of the Cambridge

Ring make it especially suitable. Current implementations carry only 18 information bits in each slot [SHAR82] which is too small for MININET. However, more adaptable versions are anticipated, which would allow a complete MININET packet with header and error checking field to be placed in a single slot.

One disadvantage of the slotted ring is the need for a supervisor station to set up and maintain the slot structure. (In the Cambridge Ring, virtue is made of necessity by placing powerful diagnostic procedures in the supervisor station [HOPP79].) In addition to reliability considerations, an important disadvantage of the need for a supervisor station is the relatively large initial installation cost of the ring, even when it contains only a few stations.

(3) Register Insertion Rings

With this technique a station can inject a message into the ring at once, provided that another message is not being relayed through the station. If the ring is busy, the station can start to transmit its own message immediately following the message being relayed. If a message is received for relaying, during the time the station is injecting its own message into the loop, the incoming message is queued in a variable length shift register, until the new message has been transmitted. Thus, the shift register has effectively been inserted into the loop, so giving this access method its name. A station cannot inject another message until it has sufficient memory space to buffer any incoming messages during transmission. Therefore, the performance of this access method is strongly influenced by the strategy used to remove the inserted shift register. Its best known implementation is the Distributed Loop Computer Network (DLCN) developed at the Ohio State University [REAM75]. In this implementation, the message is removed from the loop by the destination and the inserted register is shortened, as and when gaps occur in the incoming message stream. This method allows the possibility of hogging in a similar manner to that of the original Pierce loop. In an independent development, Hafner et al. suggested three different removal methods [HAFN74]. One of these allowed the message to complete a full revolution, so that it filled the inserted register, whereupon the register was removed from the ring. This idea has the advantage of minimal delay in each receiving station (as opposed to buffering enough of the message to be able to recognize the destination

address in DLCN) and immunity to hogging. Consequently, it was an early proposal by Wilkes for the Cambridge Ring [WILK75]. One characteristic that all register insertion techniques have in common is that, in contrast to the other access methods, the transit delay is variable but bounded by the total maximum length of all buffers in the loop. For the short, fixed-length packet MININET case, the method first proposed for the Cambridge Ring is very attractive both as far as its operating characteristics are concerned and the potential simplicity of its implementation.

The management of the DLCN is completely decentralized. However, both Hafner and Wilkes proposed the use of a supervisor station to deal with exception conditions such as corrupted address fields. A recent and apparently independent development by Hawker Siddeley Dynamics called Multilink is, in fact, a variant of the DLCN access method with the attendant danger of hogging. Its management is fully distributed even to the extent of dynamic address assignment.

(4) Centralized Ring Control

In this class of ring, information flows between a single master station and a number of slave stations. This asymmetry does not make it very suitable for use inside MININET. However, it is interesting to note that the application area of a number of implementations of this method are very close to that of MININET. The Weller loop [WELL71] and the IBM 2790 Loop [STEW70] were both intended for interconnection of computers with peripherals. The CAMAC Serial Highway [ESON76] connects a central computer to a number of CAMAC crates. A register insertion technique is used to allow short demand messages to be passed from any crate to the central computer.

All three distributed ring access methods could be used within MININET, with the appropriate message removal mechanism. It is interesting to observe that virtually all access methods, immune from hogging, require that the messages complete a full circuit of the loop, before being removed by the original transmitter. The short transit delay, access time and powerful diagnostics of the Cambridge Ring make it especially suitable, particularly if the functions of its monitor station could be distributed around the ring.

1.3.2 Bus Access Mechanisms

Interest in bus systems was triggered by the work of Metcalfe and Boggs in adapting the ALOHA packet radio system [ABRA70] to the local network environment in the Ethernet system [METC76]. This access method is called *Carrier Sense Multiple Access/Collision Detect (CSMA/CD)*. If a station has a message to transmit, it waits until the bus is silent (the "carrier" sense function), before transmitting the message. If two stations attempt to transmit at the same time, a collision occurs which corrupts the data on the bus. This corruption is sensed by one or other of the transmitters (the collision detection function), which reinforces the collision by jamming the bus to ensure that the other transmitter detects the collision. The transmitters then wait a random period of time before attempting to retransmit. In the original Ethernet, the mean of this interval was exponentially increased if multiple collisions occurred. This implies that previously unsuccessful stations are delayed longer than stations that have transmitted successfully. Thus, under heavy load conditions, Ethernet is unfair and can suffer from hogging. Note, that a higher speed version of Ethernet, but with essentialy the same collision protocol has been adopted as IEEE Standard 802.3 [IEEE84].

There have been a number of modified versions of Ethernet, which allow immediate acknowledgement of received packets [TOKO77] and more sophisticated retransmission backoff algorithms [HAIN82]. Both HYPERchannel [CHLA80] and Twentenet [NIEM84] use a CSMA bus with address based time slot mechanisms to resolve contention.

The collision danger period lasts for a length of time, following the start of transmission, equal to twice the propagation delay from one end of the bus to the other. Therefore, the bigger the average data unit size, the lower the probability of collision for the same information flow. If the message length is reduced to the same order of magnitude as the collision danger interval, the bus usage efficiency falls dramatically [METC76]. Furthermore, the access time is unbounded and the contention method is only stochastically fair under unsaturated load conditions. Consequently, this access method is not well suited to the real-time, short-packet MININET environment.

An alternative bus access mechanism uses token passing, in a similar fashion to that of the token passing ring. This provides a

guaranteed maximum access time, but still is most efficient with long messages. There have been a number of attempts to modify existing bus protocols, to allow token passing arbitration, by adding extra functionality. This can either be done at a high level within the data link controller, as in the IEEE Standard 802.4 [IEEE85A] and that proposed for PROWAY [IEC 81], or at a low level within the Physical Layer [WAWE82].

As bus lengths get longer and transmission rates get higher, so the efficiencies of the conventional bus access methods decline. A number of bus access methods have been proposed to overcome these limitations. These have the common feature that access to the bus is ordered by the physical position of the station along the bus. A burst of messages, the "data train" then makes its way from on end of the bus to the other. Most methods, like Fasnet [LIMB82], EXPRESS [TARI83] and D-Net [CHON82], use unidirectional transmitter taps. However some, like L-EXPRESS [TARI83], and Tokenet [AJMO83], use simpler bidirectional taps and timer based procedures to order transmissions.

1.3.3 True Networks

One of the first local area networks was the Spider system, developed by Fraser [FRAS74]. The topology of the network is that of a rosette consisting of slotted loops interconnected and managed by a central switch. Like MININET, it provides a connection orientated service and uses a fixed-length packet (albeit larger - 304 bits carrying 256 bits of user data). The main application area of Spider was that of resource sharing, between minicomputers, in a laboratory environment [FRAS75]. A later development at Bell Laboratories was the DATAKIT system [FRAS79]. This uses a short contention bus **within** the central node. Priority classes and the provision of local concentrator nodes are very similar to the functional distinction between Exchanges and Stations in MININET.

Pierce proposed a store-and-forward interconnection of loops to form a hierarchical topology [PIER72]. This idea has, more recently, been taken up by workers in Fujitsu as a method of interconnecting optical fibre rings [KAWA83].

Octopus is another hierarchically organized network, which has evolved at the Lawrence Livermore Laboratory [FLET73]. The network is

functionally partitioned into a number of subnetworks. There is one subnetwork for connecting terminals to the large "worker" computer, another connecting the computers to mass file storage, another for the provision of high-speed printing facilities and so on. Each subnetwork is essentially a star configuration with separate high-speed (16Mbps - 270Mbps) links between each worker computer and each subnetwork. Subnetworks are directly interconnected only through relatively low-speed 50kbps channels. In a retrospective article [WATS78], the experiences gained in the development of Octopus are discussed. The most important lesson was the need for a clear architectural model, cleanly separating the function of each layer. In Octopus, application and communication protocols had been mixed up, with different subnetworks having different protocols as a result. This caused great difficulty later, when the subnetworks were interconnected. Problems have also arisen due to the asymmetry of protocols and interfaces, which can make it difficult for worker computers to communicate directly, or remote microcomputers to directly access mass storage and high-speed output resources. A solution based on HYPERchannel, a high-speed bus developed for interconnection of large computers [THOR79], was proposed.

Note, that none of these networks was designed to serve the needs of instrumentation users. The experiences, gained from the Octopus network, support the MININET design decision to separate completely the user and network processes. MININET's flexible topology allows it to be configured to meet specific application needs. For example, very high Virtual Connection throughput and short transit delays can be provided, by the installation of a high-speed, point-to-point channel between the Stations concerned.

1.4 HISTORY OF THE PROJECT

The idea of a local area instrumentation network grew from the need to interconnect data acquisition equipment and minicomputers within the Faculty of Engineering and Science of the Polytechnic of Central London. It was soon realized that these requirements were not unique and that such a network would have more general applicability. The earliest description of the network was presented in May 1974 [MORL74]. At that time, the consensus of opinion, within the networking community, was in favour of very large packet sizes. Consequently, an investigation of the effect of packet size was undertaken [CAIN74]. Not

surprisingly, this study showed that the optimum packet size was strongly dependent on the average message size. By early 1975, the basic network service and topology, the packet size and structure, and the node functionality were established [MORL75]. However, the problem of sequential routing had not been solved and the need for congestion control was not understood. Furthermore, the management of the network, including Virtual Connection establishment, had not been defined.

Collaboration with a team from the Instituto di Automatica, University of Bologna, Italy, started in 1974. In conjunction with this team, the MININET Link Protocol (MLP), a sequential Data Link control protocol, suitable for hardware realization, was developed [NERI77]. A prototype network was constructed, consisting of 4 Stations and 2 Exchanges. A 16-bit PACE microcomputer was used to emulate the operation of the packet and channel level hardware. This resulted in an implementation so slow that it became known as the ***Snail Network!*** Largely as a result of the data link control overhead on the microprocessor, the channels operated at less than 100 packets per second. A high speed, full-duplex hardware implementation of MLP was designed and implemented [FALD78], which relieved the microprocessor of channel level responsibilities. However, the nodes themselves could not handle more than several hundred packets per second. In Bologna, the microprocessor was upgraded to an 8086, which increased the throughput to approximately 1k packets per second.

The MININET Control Protocol (MCP), a management transport protocol, was developed [MORL78] to enable Station managers to co-ordinate operations such as Virtual Connection establishment. MCP proved to be too slow for use in congestion and routing control. To handle these, a new class of message was devised. These NTAN messages travel only between nodes directly connected together through a single channel. The flow control algorithm is described in Section 2.4.1 and the routing management protocol in Chapter 4.

The original intention was to upgrade the Snail Network nodes to full speed, by adding additional hardware to implement the structure described in [CAIN78]. However, the Snail Network did reveal a number of problems of fairness and deadlock prevention, which could not be cured without a radical change in the internal node structure. Furthermore, the original architecture was designed to support only

point-to-point channels, and its mechanical realization made it difficult to add or change channel controllers. Consequently, a completely new Station design was undertaken (described in Chapter 5). This incorporates a look-ahead polling technique, which handles back pressure vectors to ensure fairness and to control congestion. The final design can handle packets at speeds approaching 1M packet per second.

A more detailed description of the technical evolution of MININET up to 1980 may be found in [NERI84].

Chapter 2

NETWORK ARCHITECTURE

2.1 THE MININET REFERENCE MODEL

With any system of reasonable complexity, it is necessary to divide it into a number of sub-systems whose function and the interconnections between them are well defined. In the case of a communication system, this usually takes the form of a hierarchical model where each layer provides a unified *service* to the layer above. In order to provide this service, the entities (sub-systems) within the layer communicate with each other by means of a *protocol*, making use of the service provided by the layer below. Layers are interconnected by means of an *interface*. Note that a protocol cannot be properly defined without the required service offered to the layer above, and the service provided by the layer below, having been previously specified.

2.1.1 Model Development

The architectural model of the Snail Network is shown in Figure 2.1. This structure is remarkably close to that of the ISO OSI Reference Model [ISO 84] despite the fact that it was conceived quite independently of the ISO work. The "Channel Level" provided a Layer 2, Data Link Service. Within each channel controller was a line conditioning board, whose function roughly corresponded to that of the OSI Physical Layer. The Network Layer within MININET was divided into two sublayers. The ***Packet Switching Sublayer (Layer 3P)*** supplies a Packet Delivery Service, which maintains logical paths between source and destination nodes. The ***Virtual Connection Sublayer (Layer 3V)*** provides the user with a word-stream Network Service. Splitting the Network Layer into two sublayers allows the Virtual Connection state information to be retained only in the end nodes. This facilitates connection establishment and dynamic rerouting, as no buffers are specifically reserved for individual Virtual Connections in the relay nodes.

This model very conveniently coincided with the major architectural

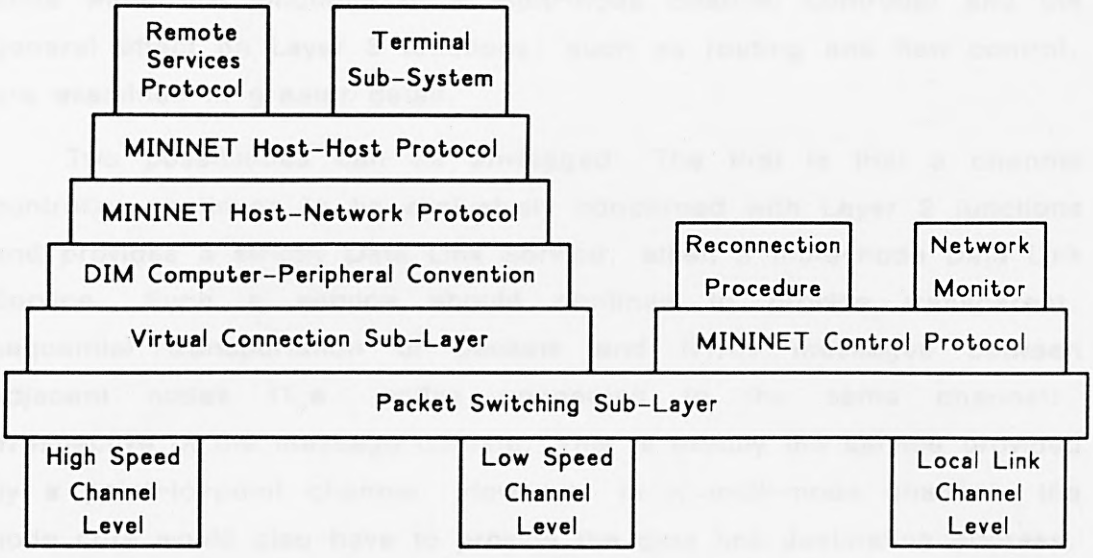


Figure 2.1: Snail Network Architectural Model

divisions within the network. Each node consists of a ***core***, concerned with Layer 3 operations, surrounded by channel controllers providing a transparent, sequential Layer 2 Service. In addition, each node has a separate management processor. The two Layer 3 sublayers are handled primarily by the different types of node. The Exchanges handle the Packet Switching Sublayer while the Stations are mainly concerned with the Virtual Connection Sublayer.

While this model was basically sound, there were a number of errors and omissions in the original concept. The errors were largely representational in nature, particularly in the depiction of the role of user management functions as a "Host-Network Protocol". The omissions were more serious. The communication requirements of the routing and congestion control algorithms entailed the introduction of NTAN messages, which did not operate through the usual management transport service. Thus, the position of routing and flow control entities within the model needed to be clarified. The original MININET conception did not include multi-node channels (i.e. rings or buses). In fact, the ISO OSI Reference Model also does not really include multi-node data links except for one or two acknowledgements to their existence. Superficially, it appears a relatively simple matter to incorporate multi-node channels into the MININET model and architecture, as the ring and bus protocol would be entirely handled within the channel

controller connecting the node to the ring or bus. However, problems arise when the functions of a multi-node channel controller and the general effect on Layer 3 functions, such as routing and flow control, are examined in greater detail.

Two possibilities can be envisaged. The first is that a channel controller continues to be exclusively concerned with Layer 2 functions and provides a strictly Data Link Service, albeit a multi-node Data Link Service. Such a service should continue to provide transparent, sequential transportation of packets and NTAN messages between adjacent nodes (i.e. nodes connected to the same channel), irrespective of the message content. This is exactly the service provided by a point-to-point channel. However, in a multi-node channel, the node core would also have to provide the data link destination address, which may or may not be the same as the Network Layer identifier of the receiving node. Similarly, when receiving an NTAN message, the channel must also supply the data link source address to enable the core to identify the transmitter. For all practical purposes, the multi-node channel would be treated as a bundle of point-to-point communication paths tied together into the same channel.

The other possibility is to delegate as many as possible of these added burdens to the multi-node channel controller. As far as packet-by-packet routing is concerned, the node core would route a packet to the correct channel controller, whereupon the channel controller routes the packet to the appropriate adjacent node. In addition, some broadcasting and pre-processing of NTAN messages could be done by the multi-node channel controllers. These functions are indisputably Layer 3 functions, since they are concerned with network addresses and they manipulate NTAN messages. Therefore, the adoption of the second option leads to certain Network Layer entities being placed in ring and bus controllers. This introduces the concept of a further Network sublayer - the ***Multi-Node Channel Sublayer (Layer 3M)***. The extra functionality required to handle multi-node channels is almost completely placed in their controllers leaving the node core virtually unchanged. Furthermore, the simpler point-to-point channel controllers are not burdened with irrelevant jobs, as the secondary routing function and the NTAN message processing reduce to that of a transparent pipe in the case of a two-node channel. Thus, Layer 3M becomes a null layer for point-to-point channels. This is illustrated in

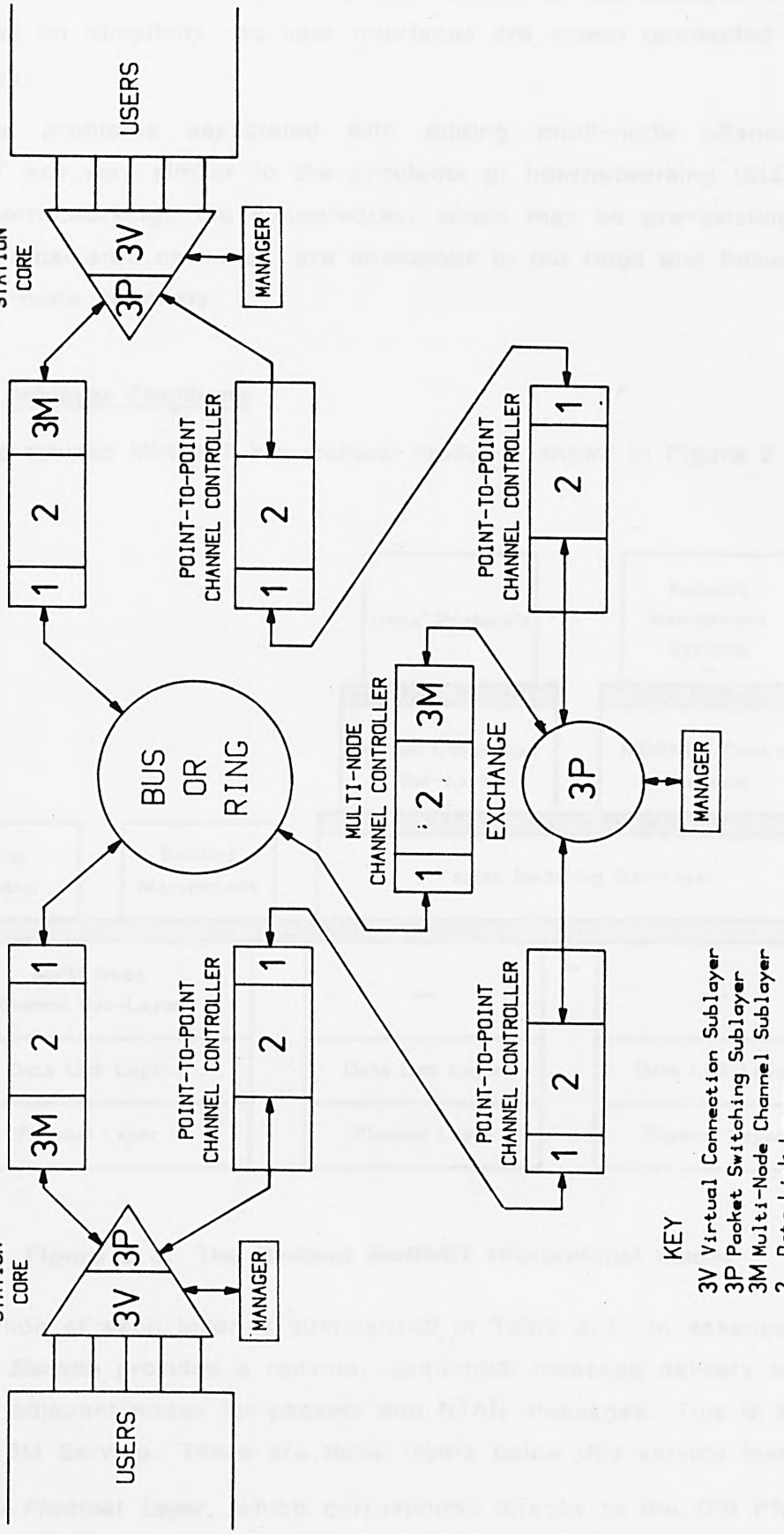


Figure 2.2: Location of Layer Entities in MININET

Figure 2.2, which shows the physical location of the sublayer entities. Note that for simplicity, no user interfaces are shown connected to the Exchange.

The problems associated with adding multi-node channels to MININET are very similar to the problems of internetworking [SLOM83]. With internetworking, the subnetworks, which may be pre-existing wide area or local area networks, are analogous to the rings and buses used as multi-node channels.

2.1.2 Sublayer Functions

The revised MININET hierarchical model is shown in Figure 2.3 and

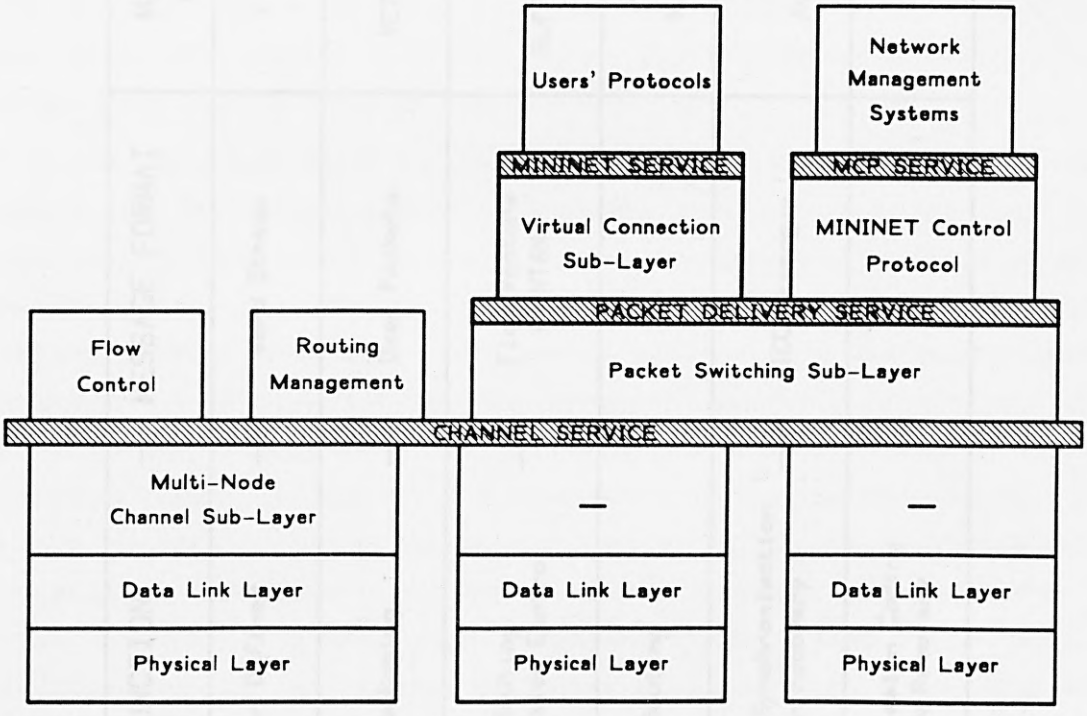


Figure 2.3: The Revised MININET Hierarchical Model

the function of each layer is summarized in Table 2.1. In essence, the ***Channel Service*** provides a reliable, sequential message delivery service between adjacent nodes for packets and NTAN messages. This is strictly a Layer 3M Service. There are three layers below this service level.

The ***Physical Layer***, which corresponds directly to the OSI Physical Layer, is responsible for transmission coding and channel symbol synchronization. The service provided by this sublayer and the interface

LAYER		FUNCTION	— MESSAGE FORMAT	MANAGEMENT FUNCTION	— MESSAGE FORMAT
	Users	User Defined	— Word Stream		
3V	Virtual Connection	Packaging	— User Packets	VCT Maintenance	— MCP Packets
3P	Packet Switching	Routing Congestion Control	— Flow Vectors (B-NTAN)	Routing Buffer allocation	— S-NTAN Messages
3M	Multi-Node Channel	Routing		Flow Vector Multiplexing	
2	Data Link	Envelope Synchronisation Error Recovery	— ICC Messages	Channel Availability	
1	Physical	Transmission Coding Clock Recovery			

Table 2.1: MININET Layer Functionality

between it and the Data Link Layer, is not specified in detail. This allows maximum freedom to tailor the network's use of whatever physical medium is most suited to an environment or application, and to allow the exploitation of any further developments in data transmission technologies. However, a de facto standard has emerged in implementation, which is suitable for high-speed, point-to-point, serial links. It allows a variety of physical media (e.g. coaxial cable or optical fibre), modulation and encoding schemes to be used with the same data link controller. This implementation has the interesting characteristic that the size of the Physical-SDU is the conventional 1 bit, while the Physical-IDU is 4 bits wide. This was done so that the Data Link Layer circuitry need only operate at one quarter of the bit rate of the physical channel. Allowing the size of the SDU to remain at one bit means that the Physical Layer receiver is not required to frame the incoming data stream to the original 4-bit boundary. Synchronization to data units larger than one bit is left to the Data Link Layer.

The ***Data Link Layer*** (corresponding directly with OSI Layer 2) is responsible for error recovery procedures that guarantee no loss of sequentiality. This would normally involve wrapping the message in an ***envelope***, which includes a header containing channel control information and some sort of checksum tail protecting the header and message. A primary task of the receiving entity, in this layer, is envelope synchronization - i.e delimiting the start and end of each envelope (block framing). The Data-Link-PDU would normally contain a single packet or NTAN message. Segmenting or blocking may be necessary if the format of the Data-Link-PDU is already fixed. However, since the SDU is already small, segmenting would tend to be rather inefficient as far as channel utilization is concerned. If blocking is performed to improve channel efficiency, messages should not be delayed excessively in order to fill a PDU. The Data Link must preserve the identity and distinction between packets and NTAN messages. The management of the Data Link includes initialization, maintenance of performance statistics, recovery from major failure and in the case of multi-node channels, the identification of other stations on the ring or bus. An important management function is to monitor continually the channel operation and decide when it is available for use. The Data Link Service, per se, is not of any explicit concern except for its effect on the Channel Service.

The primary function of the Multi-Node Channel Sublayer (Layer 3M) is to route a packet to the appropriate adjacent node. This involves mapping the destination node address of the packet into a data link address and passing this to the Data Link Layer. Note that this is a many-to-one mapping, as there may well be further hops before the packet finally reaches its destination. It is **not** the function of this sublayer to determine which is the "correct" next hop. This information is loaded into the channel controller by the routing management entity within the node core. The transmitter entities, within this sublayer, also provide a broadcast service for the transmission of certain NTAN messages, including flow vectors, to all other nodes connected to the same channel. Upon receipt of a flow vector from an adjacent node, the receiving entities must compute a new composite vector, incorporating information from the other nodes connected to the channel, and pass this to the node core.

The Packet Switching Sublayer (Layer 3P) provides the ***Packet Delivery Service*** which undertakes to transport packets reliably and sequentially from source node to destination node. It does this with the aid of the routing management and the buffer allocation algorithms. The bulk of the Exchange core is concerned with this sublayer (Figure 2.2). The main objectives of the 3P entities, within an Exchange, are to choose the correct output channel for a packet (routing), and not to allow the Exchange packet buffers to overflow (congestion control). Since a Station never performs the relay function of forwarding a packet from another node to some other node, it does not have a great deal of functionality within this sublayer. However, it must take part in the distributed routing management algorithm, so that the presence of the Station is known to the rest of the network and to enable it to choose the best output channel for packets originating at the Station. It must also take part in the congestion control protocol as discussed in Section 2.4.1.

The Virtual Connection Sublayer (Layer 3V) provides a connection based service to the network ports and so to the network users. Since this is the only service which is visible externally to the users, it is called the ***MININET Service.*** In the original MININET design, the only type of interface into the network was a DIM port. Any other type of interface, such as IEC-625 or the V24 asynchronous line interface, required an interface translator external to the network. This translator

was considered by the network to be part of the user device. Thus, the Network Service boundary was the DIM port interface. While architecturally and administratively sound, this approach was not always cost effective because of the frequent need for interface translators in addition to the ports. The alternative now adopted is to allow ports to have different types of interfaces. However, unlike DIM, interface standards such as the IEC-625 instrumentation bus [IEC 79] were not designed for use with a local area network. Consequently, their port design is more complex than that of a DIM port and a protocol has to exist between the ports themselves in order to maintain the "fiction" of a direct local connection between devices. This is in contrast to a DIM port, which acts as a straightforward interface into the network. In order to allow devices connected though a DIM port to communicate with devices connected though some other type of port, the other types of port must use a protocol compatible with the DIM Computer-Peripheral Convention described in Section 3.3. The existence of a protocol actually between ports, as entities in their own right, means that a new sublayer, in the OSI sense, has been created. This layer could be thought of as an interface sublayer at the top of the network. Such a model is not entirely satisfactory, as the Network Service would have to be defined in terms of the port design. There would no longer be a single service specification but many depending on the number of port designs. Problems would also arise with DIM devices connected though a DIM port to a device via a non-DIM port, because the DIM device (outside the network) would be communicating on a peer-peer basis with the non-DIM port (inside the network). All these difficulties are avoided by placing the MININET service boundary between the ports and the Station core.

The Packet Delivery Service is used by MCP [MORL78] to provide a highly secure, half-duplex, connection oriented Transport Service (OSI Layer 4) for the management entities within each node.

2.1.3 Relationship to the ISO OSI Reference Model

As far as layer identification and boundaries are concerned, the MININET and OSI models are very similar. However, in other aspects there are a number of important differences. The most fundamental of these is that the OSI model is an abstract communication model, while the MININET model is an implementation model. As a result, the OSI

model is not concerned with physical or administrative divisions. On the other hand, the MININET model is designed to provide an architectural framework for a specific implementation. As such, it is much more concerned with the physical divisions required to meet the performance specifications for the network. Boundaries between the layers of the model and the physical divisions in the nodes are expected to coincide, as shown in Figure 2.2.

The OSI Reference Model, in its original form [ISO 84], used only one type of connection-oriented service. Although the model has been extended to include connectionless services [ISO 87], with a variety of connectionless service types being proposed [VISS85], there appears to be no sign of any effort to include other types of connection-oriented service in the model. Figure 2.4 illustrates the four-phase sequence of

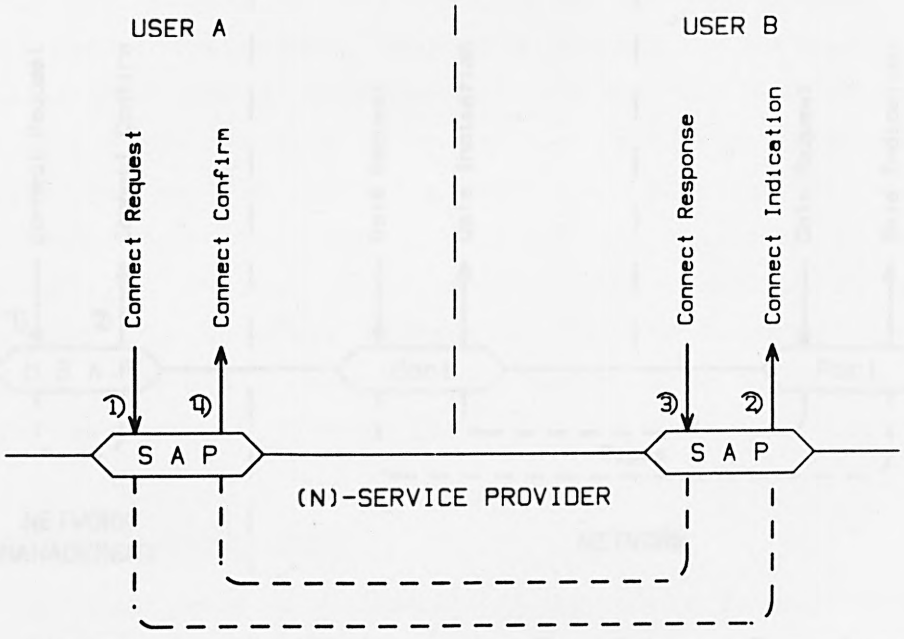


Figure 2.4: OSI Connection Establishment Procedure

service primitives required to establish an (N)-connection, through a pair of (N)-Service-Access-Points (SAPs), as defined in the OSI reference model. Note that, **both** the (N+1)-entities are involved in the connection establishment and must be actively present for the connection to exist. This has been criticised, by proponents of connectionless services [VISS85], as being unnecessarily complex. Furthermore, it requires a fairly high degree of network awareness on the part of the service user during the connection establishment phase. In the case of

MININET, this is undesirable as it compromises the requirement for transparency (Section 1.2.2).

An alternative approach has been used in MININET. Connection establishment is treated as a management function. A user management entity requests, via a ***management-service-access-point (MSAP)***, a Virtual Connection to be established between two ports of the network. Network management entities then attempt to set up the connection, whereupon the user management entity is informed of the success, or otherwise, of the connection establishment. This approach requires only two connection service primitives, as shown in Figure 2.5. The devices

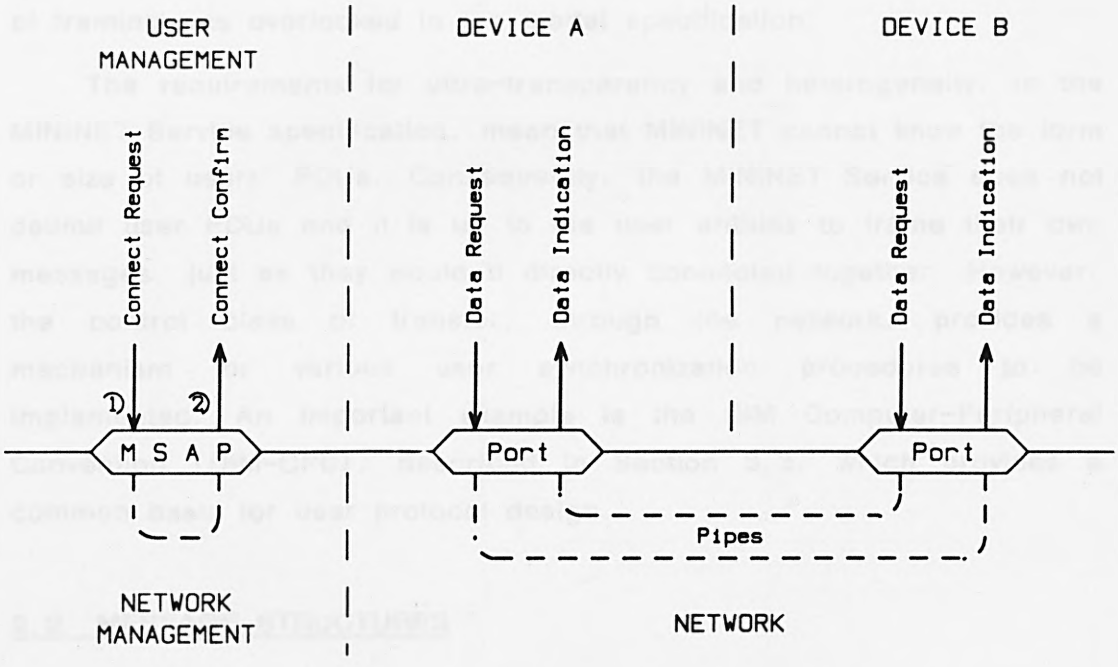


Figure 2.5: MININET Connection Establishment Procedure

connected to these ports are not involved in the connection establishment procedure. Consequently, the availability or even existence of the connected devices is not established by the Network Connection - only the existence of pipes between the connected ports. Device readiness can only be ascertained by the peer-peer protocol between the connected devices just as if they had been physically connected directly together. The actual realization of two types of MSAP are described in Section 2.5.1.

A service function that is not explicitly described in the OSI Reference Model is PDU delimitation. The general concept of breaking an (N)-PDU into a number of (N-1)-SDUs is not described by the ISO standard, on the implicit assumption that an (N-1)-SDU will contain at least one (N)-PDU (although it may contain more, if the (N)-entity performs concatenation). However, in the specific description of the Physical Layer, the usual SDU is envisaged to be one bit - clearly much smaller than the Data Link PDU. In practice, the job of delimiting Data Link PDUs (otherwise known as framing) may be done either within the Data Link Layer or is a service provided by the Physical Layer (e.g. through the use of special transmission codes). For the other layers, it is implicitly assumed that PDUs are delimited by the layer below, unless concatenation has been performed. Indeed, the whole important problem of framing was overlooked in the model specification.

The requirements for ultra-transparency and heterogeneity, in the MININET Service specification, mean that MININET cannot know the form or size of users' PDUs. Consequently, the MININET Service does not delimit user PDUs and it is up to the user entities to frame their own messages, just as they would if directly connected together. However, the control class of transfer, through the network, provides a mechanism for various user synchronization procedures to be implemented. An important example is the DIM Computer-Peripheral Convention (DIM-CPC), described in Section 3.3, which provides a common basis for user protocol design.

2.2 MESSAGE STRUCTURES

Four major types of messages are exchanged between entities at various levels within the network. The difference in their purpose is best revealed by the difference in their scope, as represented in Figure 2.6. These are ***packets***, ***node-to-adjacent-node (NTAN) messages***, ***intra-node-control (INC) messages***, and ***intra-channel-control (ICC) messages***.

(i) The "long distance" information carriers are packets which travel and retain their identity, from a source node to a destination node anywhere in the network, possibly through one or more relay nodes. Their scope is, therefore, network wide. There are two types of packet. A ***user packet*** contains data transferred

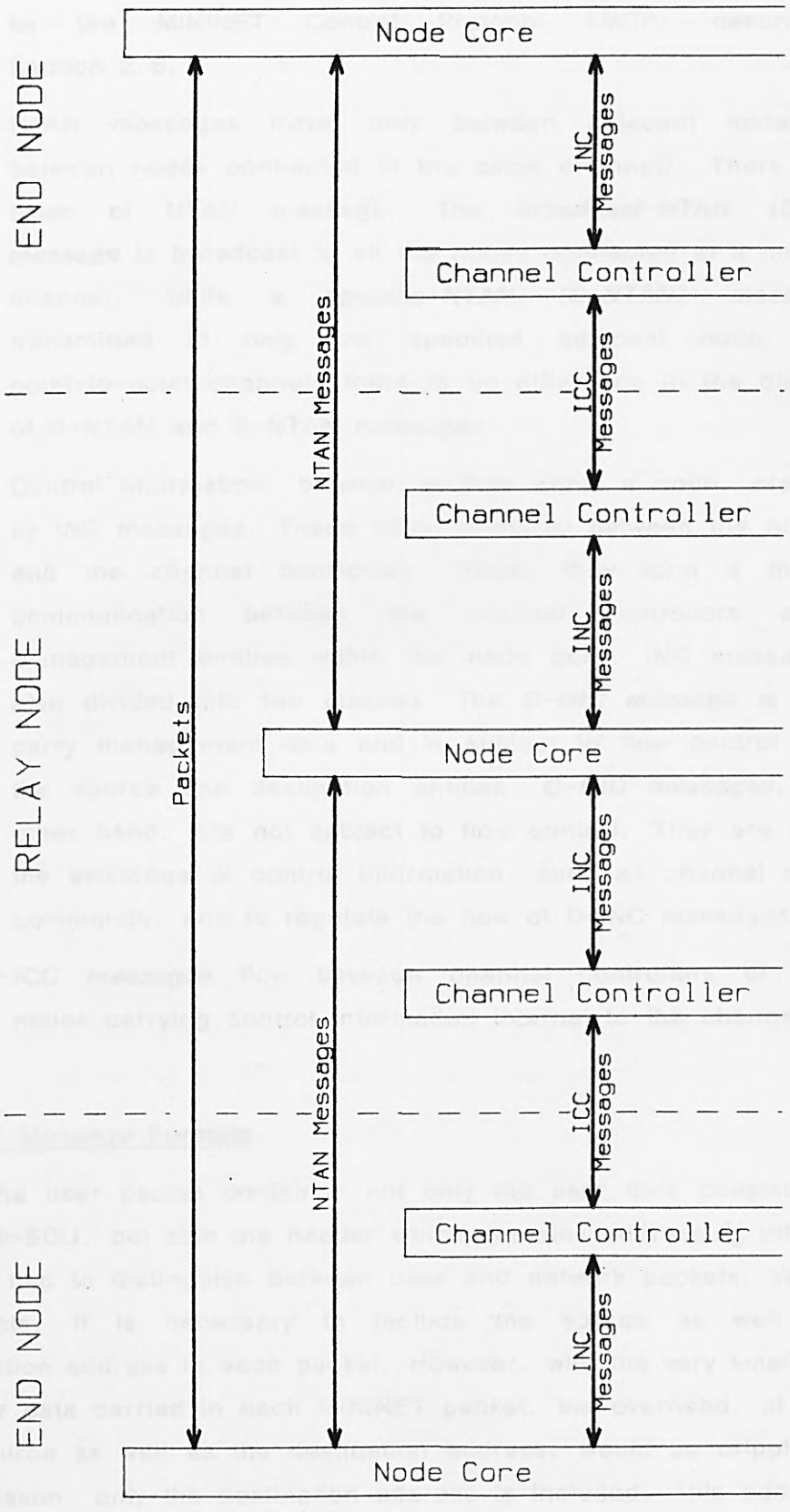


Figure 2.6: MININET Message Scope

across a network port in the source node. A ***network packet*** carries information from one node manager to another. It is used by the MININET Control Protocol (MCP) described in Section 2.6.

(ii) NTAN messages travel only between adjacent nodes (i.e. between nodes connected to the same channel). There are two types of NTAN message. The ***broadcast-NTAN (B-NTAN) message*** is broadcast to all the nodes connected to a multi-node channel, while a ***specific-NTAN (S-NTAN) message*** is transmitted to only one specified adjacent node. For a point-to-point channel, there is no difference in the distribution of B-NTAN and S-NTAN messages.

(iii) Control information, between entities within a node, are carried by INC messages. These travel primarily between the node core and the channel controllers. Thus, they form a means of communication between the channel controllers and the management entities within the node core. INC messages are also divided into two classes. The ***D-INC message*** is used to carry management data and is subject to flow control between the source and destination entities. ***C-INC messages,*** on the other hand, are not subject to flow control. They are used for the exchange of control information, such as channel status or commands, and to regulate the flow of D-INC messages.

(iv) ***ICC messages*** flow between channel controllers of adjacent nodes carrying control information internal to the channels.

2.2.1 Message Formats

The user packet contains, not only the user data consisting of a network-SDU, but also the header which contains addressing information and a flag to distinguish between user and network packets. With most protocols, it is necessary to include the source as well as the destination address in each packet. However, with the very small amount of user data carried in each MININET packet, the overhead, of carrying the source as well as the destination address, would be crippling. For this reason, only the destination address is included. This address has two components: the destination node and the destination port. No source address is required, because a port can be virtually connected to

only **one** other port at any one time. The size of the address fields is a compromise between the desirability of minimizing the size of the packet header and maximizing the potential maximum number of nodes in the network and ports in a node. A 6-bit length was selected for both the node and port address fields. Therefore, the maximum number of nodes in the network and ports in a node are restricted to 64 in both cases. The small packet size also precluded any sequence field, as discussed in Section 1.2.3. The inclusion of two parity check digits, to fulfil the requirements described in Section 1.2.4, results in a total packet size of 32 bits.

The overall formats of packets, NTAN messages and INC messages are shown in Figure 2.7. Since ICC messages are completely internal to the various types of channel, their formats do not require standardization across the network. In transit within a node, these different types of message are distinguished by means of a ***message type field*** as shown. Network packets are distinguished from user packets by means of the ***network message flag (NMF)***. In addition to the NMF, the address section of an user packet (Figure 2.7a) contains the ***destination node address (DNA)***, the ***destination port address (DPA)*** and a check bit (AFP) which makes the parity of the address section odd. The data fields of a user packet consist of a 16-bit data field, the ***data class flag (DCF)*** and an odd-parity check bit (DFP). These 18 bits of the data section form the single Network-SDU/IDU, as discussed in Section 1.2.2. The network packets (Figure 2.7b) have a similar structure to the user packets except that, instead of a DPA, it has a ***source node address (SNA)*** field and, instead of a DCF, it includes a modulo-2 ***sequence number (SQN)***. These are used by MCP.

The ***adjacent node address*** (NODE) field (Figure 2.7c), accompanying a NTAN message when travelling from a multi-node channel controller to the node core, is generated by the channel controller and identifies the adjacent node that originated the message. The NODE field, accompanying a S-NTAN message when travelling from the core to a multi-node channel controller, identifies the adjacent node to which the S-NTAN message should be sent. For B-NTAN messages travelling from the core to the channel, the content of the NODE field is immaterial. This is also true for all NTAN messages travelling to or from a point-to-point channel since there is only one other node connected to that channel. This address is protected by an odd-parity bit (ANP).

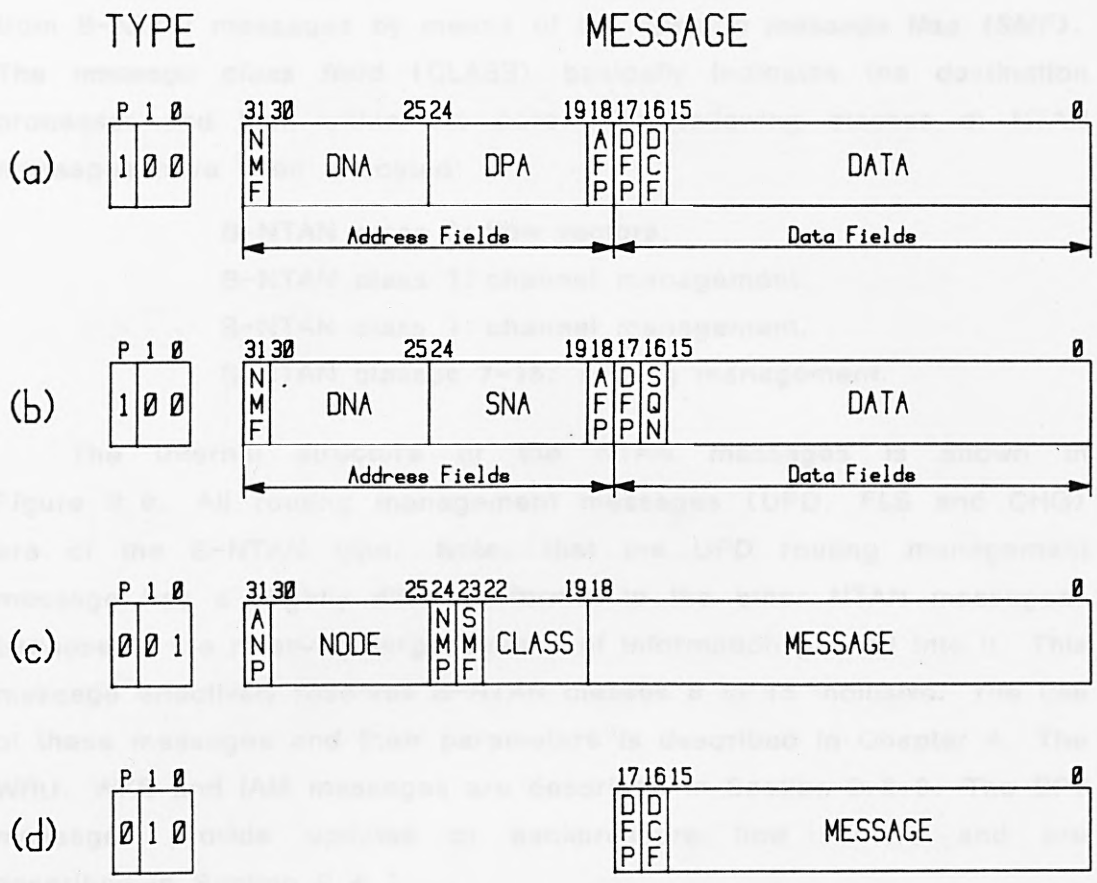


Figure 2.7: MININET Message Format

(a) User packet (NMF=0); (b) Network packet (NMF=1); (c) NTAN message; (d) INC message.

Key:

NMF = Network message flag
DNA = Destination node address
DPA = Destination port address
SNA = Source node address
AFP = Address fields parity
DFP = Data fields parity
DCF = Data class flag
SQN = Sequence number
ANP = Adjacent node address parity
NODE = Adjacent Node address
NMP = NTAN message parity
SMF = Specific message flag
(0 = B-NTAN, 1 = S-NTAN)
CLASS = Message class

Note that this must be calculated correctly, even when the adjacent node address is not of any significance. S-NTAN messages are distinguished from B-NTAN messages by means of the ***specific message flag (SMF)***. The ***message class field*** (CLASS) basically indicates the destination processor and task within the core. The following classes of NTAN messages have been allocated:

B-NTAN class 0: flow vectors;
B-NTAN class 1: channel management;
S-NTAN class 1: channel management;
S-NTAN classes 7-15: routing management.

The internal structure of the NTAN messages is shown in Figure 2.8. All routing management messages (UPD, FLS and CHG) are of the S-NTAN type. Note, that the UPD routing management message has a slightly different format to the other NTAN messages, because of the relatively large amount of information packed into it. This message effectively reserves S-NTAN classes 8 to 15 inclusive. The use of these messages and their parameters is described in Chapter 4. The WRU, WKE and IAM messages are described in Section 2.3.3. The BPV messages provide updates of backpressure flow vectors and are described in Section 2.4.1.

The INC message has a similar format (Figure 2.7d) to the data section of a user packet.

2.2.2 Packet Size Considerations

The size of the user data section of the packet (Network-PDU) has been constrained to one Network-IDU by the transparency requirements of the network (Section 1.2.2). The size of the packet header is determined primarily by the size of the node and port address space as discussed in the previous section. However, it is useful to consider the effect, of this relatively small fixed packet size, on the overall efficiency of the packet transmission and of the storage in network relay nodes. This has been discussed at length in [CAIN74].

Let M be the message (i.e. the user-PDU) length, D be the (fixed) size of the packet's data field and H be the size of the packet's header (and trailer, if any). Note that M is, in general, a random variable whose distribution is a characteristic of the user traffic, while D

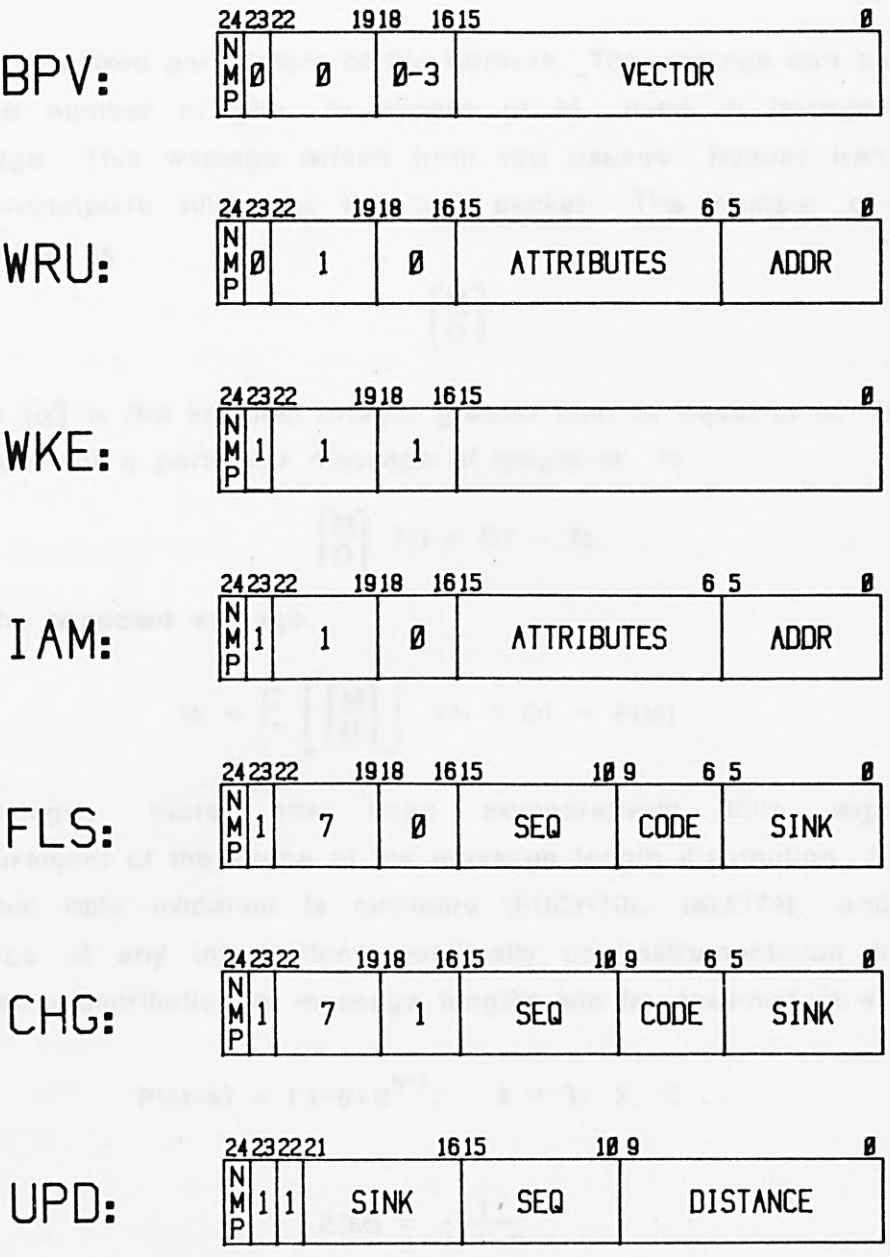


Figure 2.8: NTAN Message Structure

BPV = backpressure flow vector update;
NMF = NTAN message parity (odd);
VECTOR = 16-bit segment of new flow vector.

WRU = "Who are you?" - adjacent node identification request;
ATTRIBUTES = source node characteristics;
ADDR = address of source node.

WKE = "Wake" - link rehabilitation request.

IAM = "I am" - node identification message.

FLS = routing flush message;
SEQ = update cycle sequence number;
CODE = sub-function code;
SINK = address of root node.

CHG = routing connectivity change message.

UPD = routing distance update;
DISTANCE = distance of source node from the SINK.

and H are fixed parameters of the network. The wastage can be defined as the number of bits, in excess of M, used in transporting the message. This wastage arises from two causes: header transmission and incomplete filling of the last packet. The number of packets transmitted is

$$\left\lceil \frac{M}{D} \right\rceil$$

where $\lceil \alpha \rceil$ is the smallest integer greater than or equal to α. Thus, the wastage, for a particular message of length M, is

$$\left\lceil \frac{M}{D} \right\rceil (H + D) - M$$

and the **expected** wastage,

$$W = E\left[\left\lceil \frac{M}{D} \right\rceil \right] (H + D) - E[M] \qquad (2:1)$$

Surprisingly, there has been comparatively little experimental measurement of the shape of the message length distribution. Supported by what little evidence is available [FUCH70], [KLEI74], and in the absence of any information specifically on instrumentation traffic, a geometric distribution of message lengths will be assumed, i.e.

$$P\{M=k\} = (1-\theta)\theta^{k-1}; \quad k = 1, 2, 3... \qquad (2:2)$$

and

$$E[M] = \frac{1}{1 - \theta} \qquad (2:3)$$

Then

$$P\left\{ \frac{M}{D} = j \right\} = P\{ (j-1)D < M \leq jD \}$$

$$= (1-\theta) \sum_{k=(j-1)D+1}^{jD} \theta^{k-1} = (1-\theta^D)(\theta^D)^{j-1} \qquad (2:4)$$

This is, again, a geometric distribution with a new parameter, θ^D, substituted for θ. The expected value is, therefore,

$$E\left[\left\lceil \frac{M}{D} \right\rceil \right] = \frac{1}{(1 - \theta^D)} \qquad (2:5)$$

Thus, the expected wastage becomes

$$W = \frac{H + D}{1 - \theta^D} - \frac{1}{1 - \theta} \qquad (2:6)$$

where $$\theta = 1 - 1/E[M]. \qquad (2:7)$$

If this function is plotted against increasing E[M], it initially falls from its starting value of H+D−1 before rising again to approach asymptotically, from above, the line

$$\frac{H}{D} E[M] + \frac{(H+D)(D-1)}{2D} \qquad (2:8)$$

This asymptotic bound can be obtained by taking the limit, as E[M] tends to infinity, of

$$W - \frac{H}{D} E[M]$$

Unfortunately, it is not possible to obtain an expression for the wastage minimum in closed form. Therefore, the minimum wastage can only be found using numerical techniques.

Strictly speaking, the size of the user data field in the MININET packet is 17 bits, or even 18 bits if the data fields parity bit is included. However in practice, a large block of user data would be segmented into 16-bit segments, with the data class flag used to indicate that data is being carried. The control class transfers are used, among other functions, to delimit the user messages. Thus, as far as buffer utilization within the Exchanges is concerned, the header overhead can be taken as 16 bits, with a data field length of 16 bits. Using (2:6), the consequent expected wastage per message, as a function of average message length, is plotted as the solid line in Figure 2.9 with H = 16 bits and D = 16 bits. It can be seen that, as the mean length of the messages increase, the expected wastage falls to a minimum, before increasing towards the asymptotic line, E[M] + 15, as predicted by (2:8). The minimum wastage occurs with a mean message length of only 6 bits. As far as buffer wastage is concerned, this is the optimum average message length for the given values of D and H. However, this does **not** imply that 16 bits is the optimum data field size for a mean message length of 6 bits. To find this optimum value, it is necessary to investigate expected wastage as a function of D. This is plotted, again using (2:6), for mean message lengths of 6, 12 and 24 bits, in Figure 2.10. This shows that, in fact, 11 bits is the optimum data field

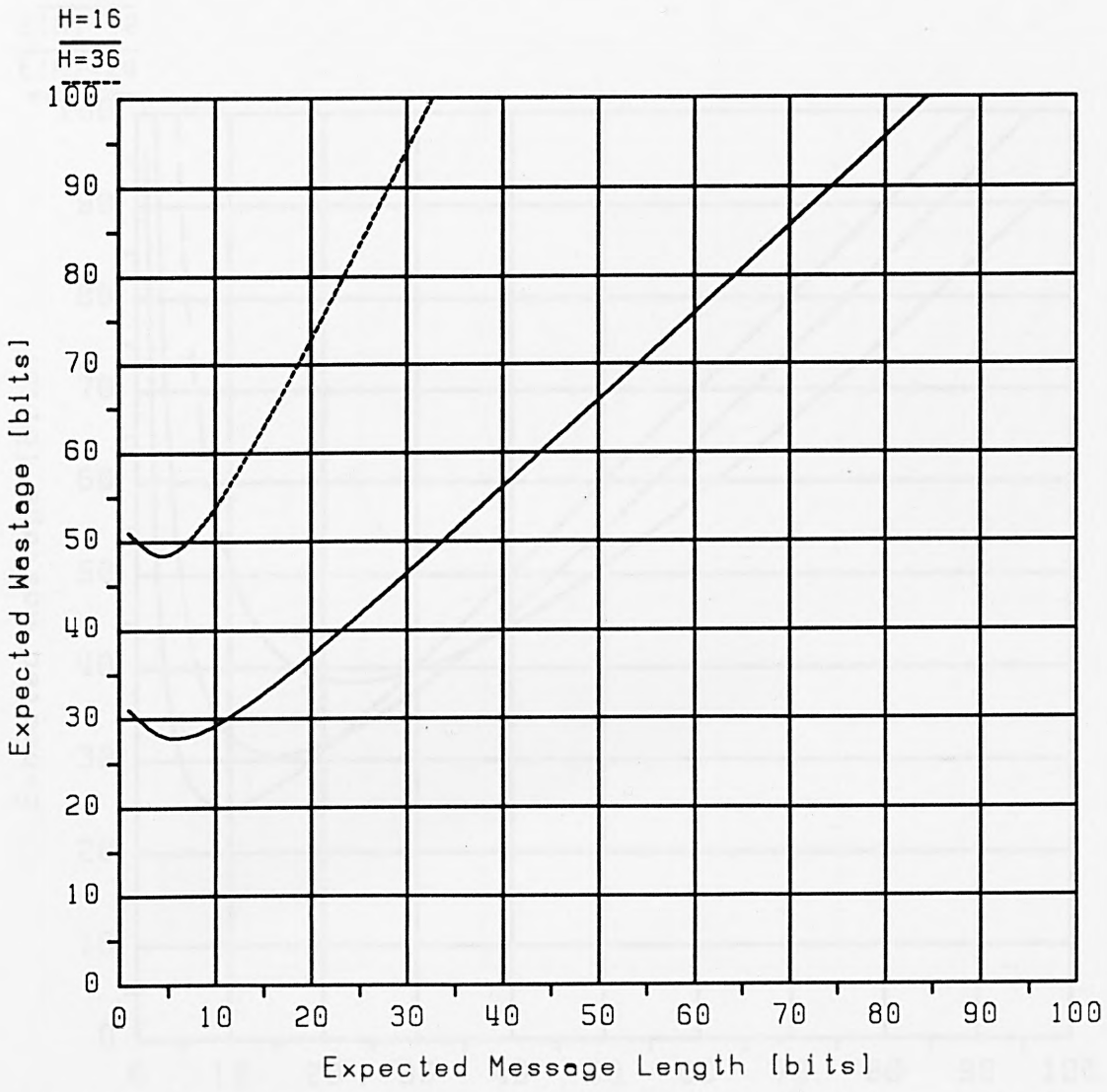


Figure 2.9: Expected Wastage Versus Average Message Length (D=16)

size for E[M]=6 bits. Furthermore, Figure 2.10 implies that D=16 bits is the optimum for mean message lengths in the order of 12 bits. The total picture can be better understood by means of Figure 2.11, which plots expected buffer wastage as a function of both data field and expected message length. Superimposed, on these wastage contours, are the loci of minimum expected wastage for a given mean message length and for a given data field size. This confirms that a data field size of 16 bits is the optimum value for mean message lengths around 13 bits. For average message lengths in the range 1 bit to 20 bits, the wastage is within 20% of this optimum value. Note that the buffer wastage considered here is due only to overheads within the **used** buffers. Wastage due to buffer utilization is additional to that estimated here and is dependent on the buffer allocation algorithm used.

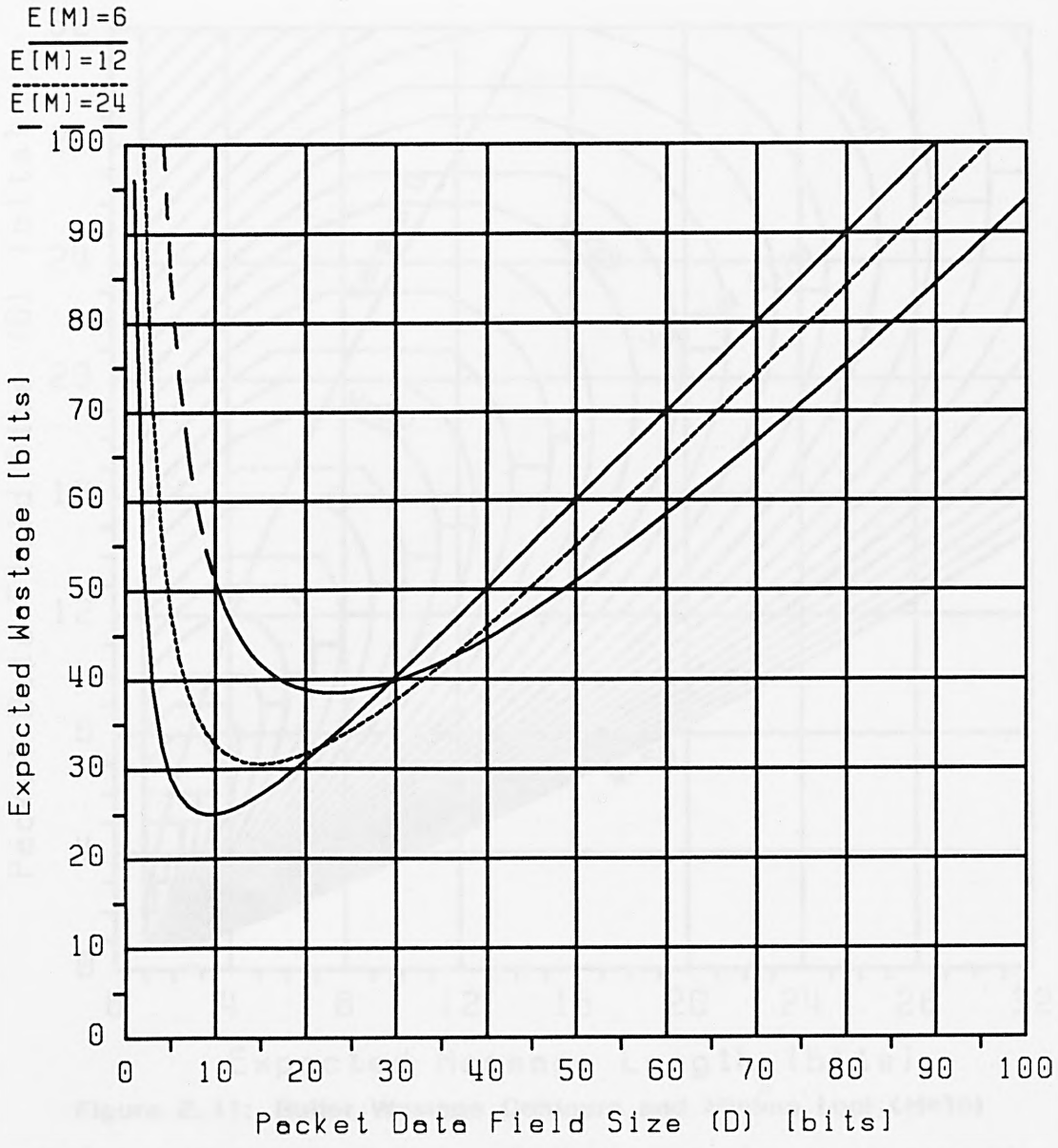


Figure 2.10: Expected Buffer Wastage Versus Data Field Size (H=16)

When the packet is actually being transmitted between nodes, it is wrapped in a channel envelope. This typically consists of a 4-bit header and a 16-bit block check field. Therefore, as far as channel wastage is concerned, H = 36 bits. The dotted line in Figure 2.9 shows the expected wastage, within the channel, as a function of average message length. Minimum wastage now occurs with an average message length of only 4.5 bits. As the average message length is increased beyond this value, the wastage increases much more rapidly than the buffer wastage case. In fact, the asymptote, as given by (2:8), is 2.25 E[M] + 24.375. Wastage contours for this header overhead are shown in Figure 2.12, together with the loci of the wastage minima against both data field size and mean message length. This shows that

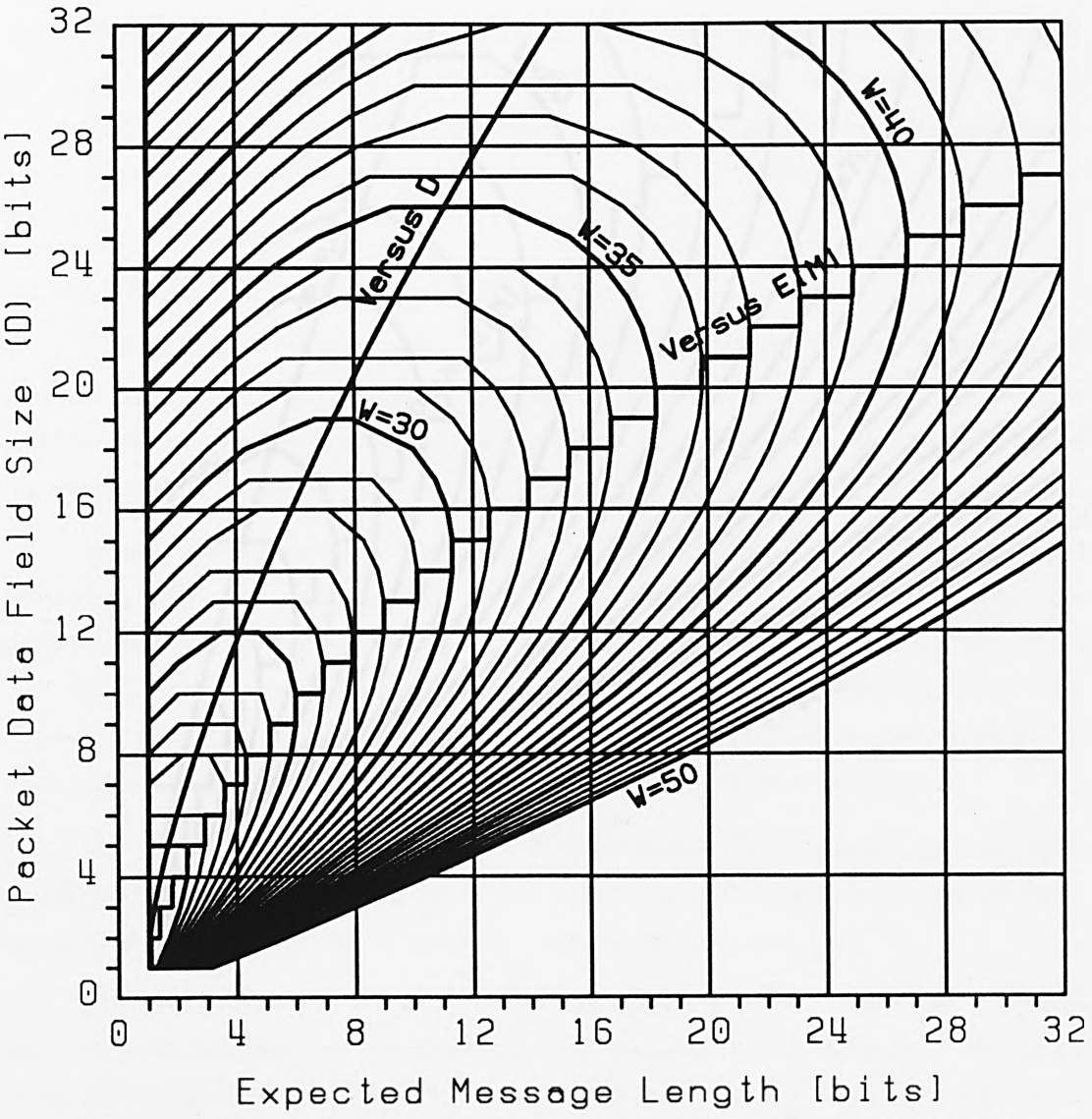


Figure 2.11: Buffer Wastage Contours and Minima Loci (H=16)

a 16-bit data field is optimum for average message lengths around 8.5 bits. The range of E[M], for which the channel wastage is within 20% of the optimum value, is 1 bit to 14.5 bits. This is quite close to that obtained in the buffer wastage case. Note, that this calculation has ignored the packet corrupting effect of noise in the channel. When the losses due to this noise are taken into account, the optimum data field size is reduced. This effect is primarily due to the lower retransmission costs of smaller packets together with the lower probability of smaller packets being damaged. However, at error rates typical of a local area network, these effects have only a very slight effect on the total wastage [CAIN74].

The results, shown in Figures 2.11 and 2.12, can be used as design curves for networks with larger message lengths, by recalibrating

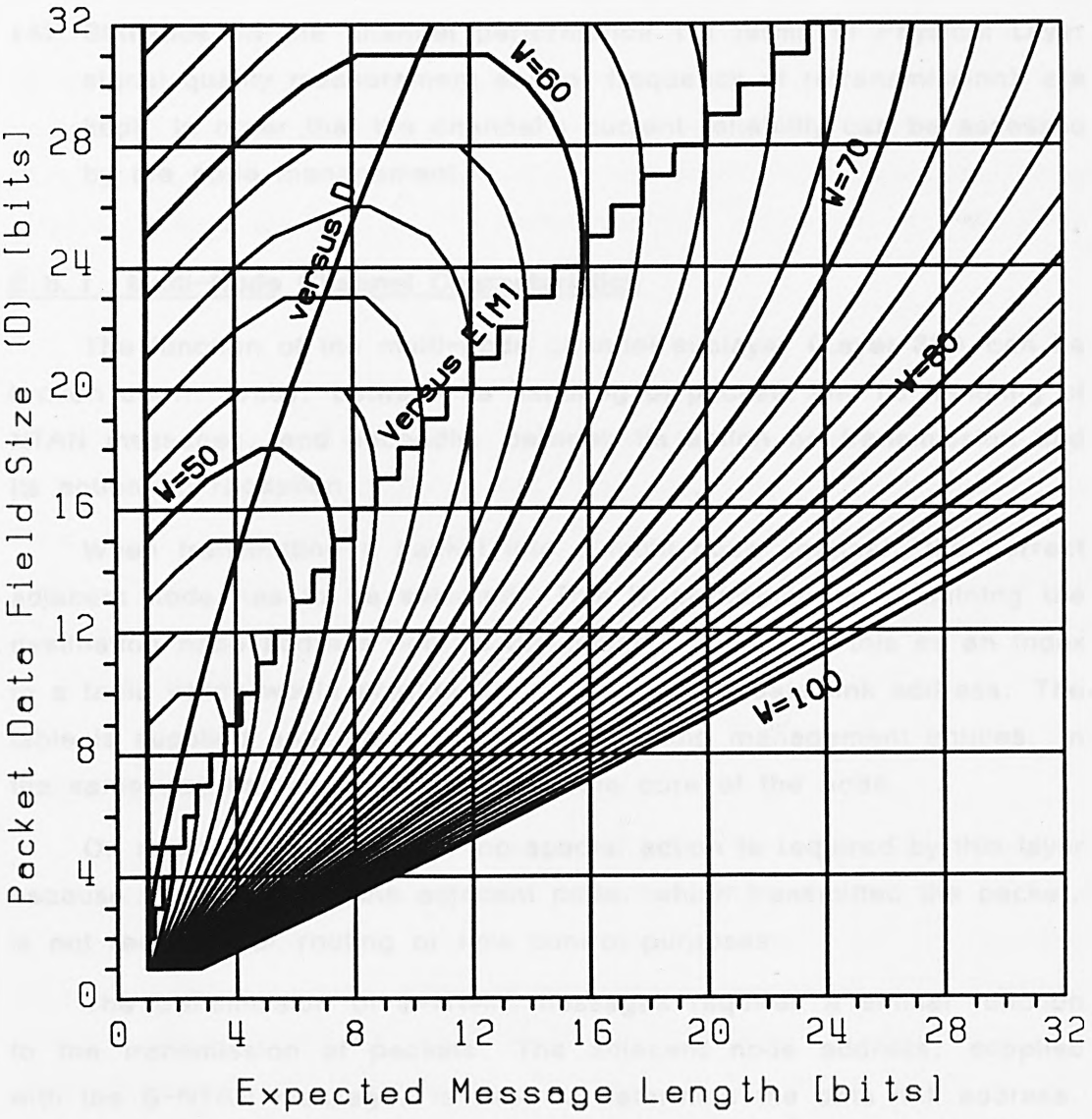


Figure 2.12: Channel Wastage Contours and Minima Loci (H=36)

recalibrating the lengths in units of bytes instead of bits.

2.3 THE CHANNEL SERVICE

The Channel Service can be summarized as follows

(1) Its primary function is to transport packets and NTAN messages sequentially between adjacent nodes.

(2) The probability, of a corrupted or duplicated packet, a corrupted or duplicated NTAN message, or a sequence error occuring, is vanishingly small.

(3) Flow control, in the sense of being able to reduce the throughput of the channel to that which can be handled by the receiving node, is provided.

(4) Statistics on the channel performance (in terms of Physical Layer signal quality measurement and/or frequency of retransmission) are kept, in order that the channel's current reliability can be assessed by the node management.

2.3.1 Multi-Node Channel Characteristics

The function of the multi-node channel sublayer (Layer 3M) can be broken down, firstly, between its handling of packets and its handling of NTAN messages, and secondly, between its action on transmission and its action on reception.

When transmitting a packet into a multi-node channel, the correct adjacent node has to be selected. This is performed by examining the destination node address field of the packet, and using this as an index to a table which would provide the corresponding data link address. The table is supplied and maintained by the routing management entities, in the same way as the routing table in the core of the node.

On reception of a packet, no special action is required by this layer because the identity of the adjacent node, which transmitted the packet, is not required for routing or flow control purposes.

The transmission of S-NTAN messages requires a similar function to the transmission of packets. The adjacent node address, supplied with the S-NTAN message, is used to determine the data link address. B-NTAN messages, on the other hand, should be transmitted to **all** extant nodes connected to that channel. If the Data Link Layer supports a broadcast service at sufficiently high quality in terms of reliability, this service can be used. Otherwise the 3M sublayer must repeatedly transmit the same message to each adjacent node in turn. Thus, one Layer 3M service primitive is mapped into several Layer 2 service primitives with identical message contents.

On reception of all types of NTAN messages, other than BPV updates, the message is passed on unchanged to the node core, together with the address of the node which originated the message. The latter is obtained by an inverse mapping from the data link source address to the corresponding node address. It is the responsibility of the channel management to construct and maintain the mapping table used for this purpose. When a BPV update is received, a new composite flow vector, associated with all packet transmissions through that channel,

must be generated. The composite vector is computed as follows. Let U_i be the flow vector most recently received from adjacent node *i*. This is the set of destinations for which node *i* has buffer space available. Also, let R_i be the set of destinations for which the transmitting entity directs packets via node *i*. The composite flow vector, **V**, is then given by

$$V = \bigcup R_i \cap U_i \qquad (2:9)$$

where the union is taken over all adjacent nodes connected to the channel. Since only 16 bits of the flow vector are updated at any one time, it is necessary to recompute only the affected quarter of the flow vector each time. The composite vector must also be recomputed whenever a routing change is made within the channel. The use made by the node core of the flow vectors is described in Section 2.4.1.

2.3.2 The MININET Link Protocol (MLP)

Conventional link protocols, such as HDLC [ISO 79], were designed for relatively large, variable-length packets. In the case of MININET, a protocol was required to handle very short packets and to maintain inherent sequency, even when packets are retransmitted. This required protocol could exploit the simplicity of the fixed-length packets to enable easy implementation in hardware. However, it had to be robust enough to withstand multiple errors.

A half-duplex protocol, used in the terminal concentrators of the NPL network [SCAN69], provided the basis for the ***MININET link protocol (MLP)*** [NERI77]. In fact, MLP can be described as an extension of the NPL protocol which makes it full-duplex, thus exploiting more fully the bandwidth of the physical channel. Another advantage of full-duplex operation is that frames can be transmitted continuously, so avoiding the need for any delimiting flags or preambles. Thus, the transmission is made fully synchronous at the frame level as well as at the symbol level. As a consequence, when there are no messages available to transmit, it is necessary to flag the unused frames as ***dummy*** envelopes. If one side of the link synchronizes its transmitter clock to its receiver clock (as is done in a regenerative repeater), then, since the packets are of fixed-length, there is a stable one-to-one correspondence between

transmitted and received envelopes at both ends of the link. This ***packet interlock*** allows each outgoing envelope to carry ***piggyback*** the acknowledgement for the last envelope received. Figure 2.13 shows an

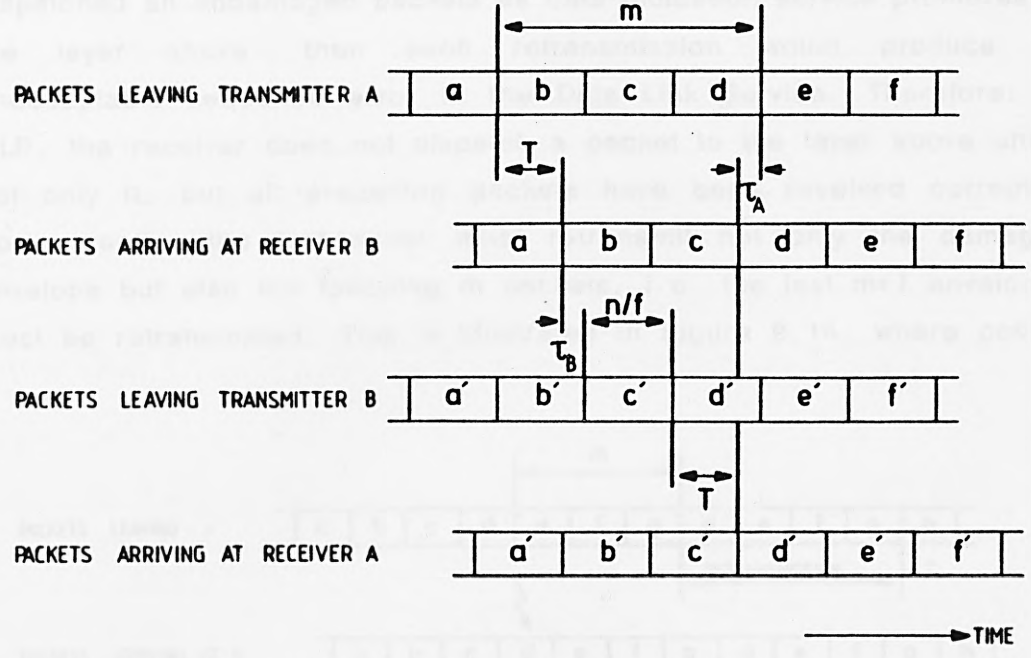


Figure 2.13: Time Placement of Interlocked Envelopes (m=3)

example of the time placement of envelopes at both ends of the channel. Let T be the one-way propagation delay between the two nodes, τ_A and τ_B be the turn-round delay between the arrival of the last symbol of an incoming packet and the start of transmission of the following return packet within nodes A and B respectively. If n is the number of transmission symbols in each envelope, and f is the Baud rate, then n/f is the time taken to transmit one envelope. Let m be the ***channel latency***, i.e. the number of subsequent envelopes transmitted before the acknowledgement of an envelope is received. From Figure 2.13,

$$m = \frac{2T + \tau_A + \tau_B + n/f}{n/f} \qquad (2:10)$$

In Figure 2.13, m=3. Note that, since $T > 0$, $\tau_A > 0$ and $\tau_B > 0$, the minimum value of m in a full-duplex channel without gaps between envelopes is 2.

If a receiver detects a damaged envelope, there will be a delay before the transmitter at the other end of the link is informed by means

of an ***error message (EM)*** carried piggyback on the interlocked envelope in the return channel. During this time, m other envelopes will have been transmitted. When the damaged envelope is then retransmitted, it will appear behind those intervening packets. If the receiver had simply dispatched all undamaged packets as data indication service primitives to the layer above, then each retransmission would produce an unacceptable sequence error in the Data Link Service. Therefore, in MLP, the receiver does not dispatch a packet to the layer above until, not only it, but all preceding packets have been received correctly. Consequently, the transmitter must retransmit not only the damaged envelope but also the following m packets, i.e. the last m+1 envelopes must be retransmitted. This is illustrated in Figure 2.14, where packet

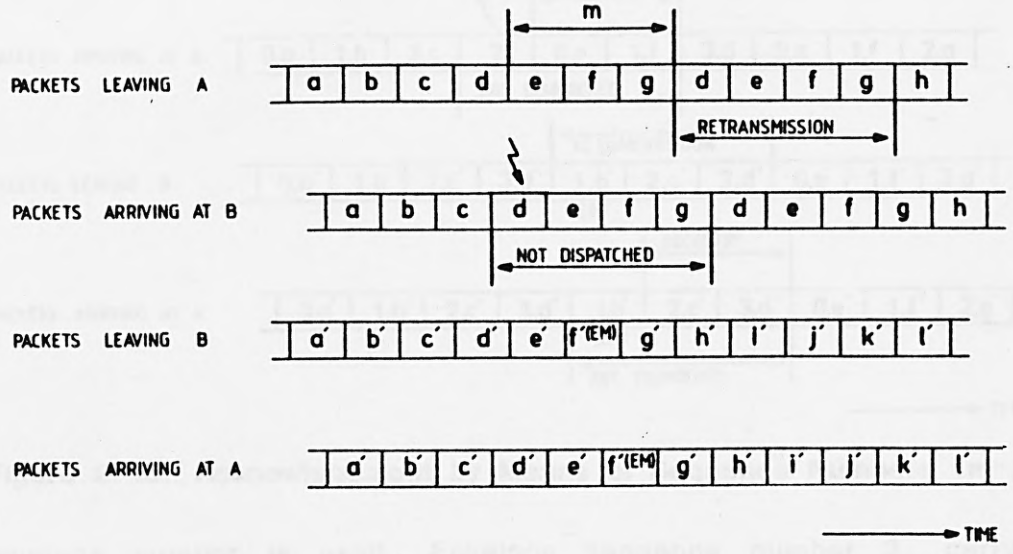


Figure 2.14: Error Recovery Maintaining Sequency (m=3)

d, sent by node A, is damaged, and node B sends an EM in envelope **f'** that is received by node A while it is transmitting packet **g**. Node A will, instead of transmitting **h**, retransmit its packets starting from **d**. The damaged envelope **d** was carrying the acknowledgement for packet **b'**. Consequently, node B does not know whether **b'** was correctly received by A. Therefore, **b'** must be retransmitted in case it was originally damaged. In order to maintain sequency in the pathway from B to A, the m packets following **b'** must also be retransmitted.

This procedure may fruitfully be used to act as the EM itself. Let each envelope carry a sequence number by which it can be identified, and let the arrival of undamaged packets be acknowledged by

transmitting their interlocked packets with a sequence number incremented by one with respect to their predecessors. Then the arrival of a damaged packet can be signalled merely by retransmitting the previous m+1 packets. The receipt of an envelope, whose sequence number is m+1 less than expected, acts as an EM, triggering the desired retransmission of the damaged packet and the following m packets. Moreover, since both directions have retransmitted m+1 packets, the packet sequences remain interlocked. Consider as an example, the case shown in Figure 2.15 where m=2 and a modulo-4

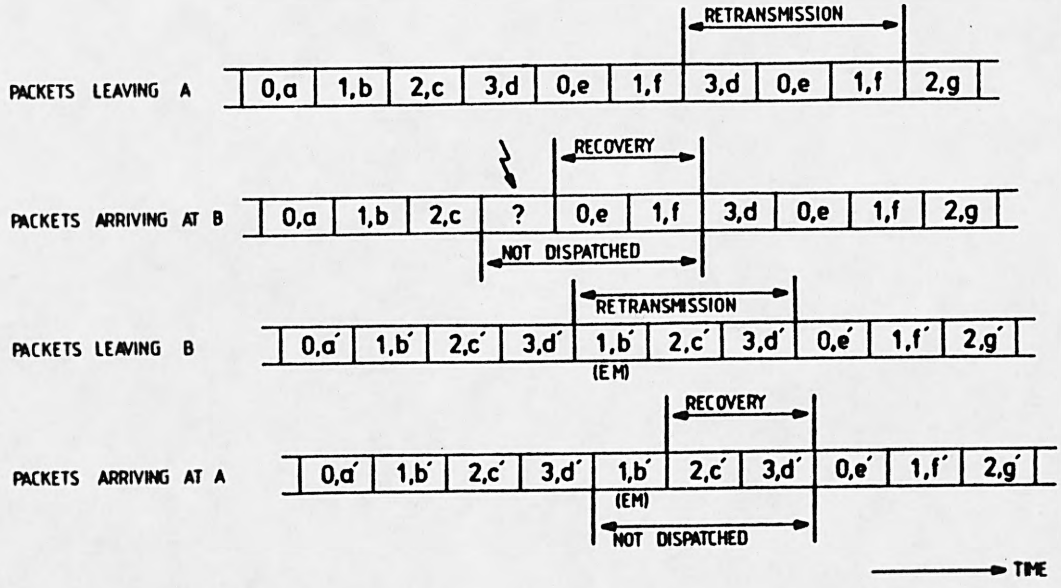


Figure 2.15: Acknowlegement by Means of Sequence Numbers (m=2)

sequence number is used. Envelope sequence number 3, carrying packet **d**, has been received damaged, by node B, which therefore, jumps back m+1 (3 in this case) packets and retransmits packet **b'** with a envelope sequence number of 1. The reception at A of a envelope number 1, instead of the expected number 0, during the transmission of packet **f**, is decoded as an EM and causes transmitter A to jump back m+1 (3), retransmitting packets **d**, **e** and **f**. Meanwhile, node B does not dispatch the m (2) packets following the damaged **d**, to avoid packet duplication and sequence errors, so that the the first packet to be accepted, once more, is the retransmission of **d**. Note that, thanks to the packet interlock, only a single sequence number is required in each envelope.

The key feature of this protocol is that the action of a transceiver, upon receipt of an EM, is identical to its action on detection of a

damaged packet: viz to retransmit the last m+1 packets. This both simplifies the protocol implementation and makes it resilient in the presence of multiple errors in either direction. In [NERI77], it is shown that MLP is reliable, even in the presence of damage to packets in the retransmission and recovery intervals (i.e. damage to an EM, damage to any of the m packets preceeding an EM or damage to any of the m packets following an EM) provided that the following rules are observed:

(a) The packet, carrying an implicit EM in its sequence number, must not be dispatched to the layer above since its packet has already been received.

(b) The m envelopes, following a damaged packet or an EM, must be ignored by the receiver as far as information carried, possible damage and sequence numbers are concerned.

(c) The (m+1)th envelope, following a damaged packet or an EM, should carry the sequence number originally expected when the damaged packet or the EM arrived. If this does not happen, it indicates that an error has occured a second time and that the recovery procedure should restart.

In order to avoid ambiguity, the sequence number must be, at least, modulo-(m+2). The half-duplex NPL protocol [SCAN69] can be obtained as a special case of MLP by setting m=0. The sequence number then becomes the modulo-2 phase bit used in that protocol.

It is possible to have a flow control mechanism, which enables traffic to be blocked in one direction, while continuing to flow in the opposite direction. This is implemented by piggybacking a "wait" flag on an envelope in the opposite direction. If a received wait flag is set, the next (interlocked) envelope is forced to be dummy, even if there is a packet waiting to be transmitted. Packets received, during the channel latency period before the wait flags take effect, must be buffered within the receiver. With this scheme, the channel appears to the layer above as a pair of **independent** queues. An alternative and simpler method of flow control is to trigger retransmissions of packets just as if they had been received damaged. This has the apparent drawback that traffic is simultaneously blocked in **both** directions. However, interlock between the two data flows is maintained. This is a positive advantage as far as the main Network Layer flow control mechanisms are concerned (Section 2.4.1), because the channel latency is constant. With

independent channel flow control, a BPV could be blocked in one direction, while packets it should be stopping are able to flow in the other direction.

In normal operation, the protocol does not distinguish between the two nodes. However, during initialization, it is necessary to designate one transceiver as the ***master*** and the other as the ***slave*** in order to establish envelope synchronization. The framing procedure is shown in Figure 2.16. Initially the master continually transmits a specially coded ***synchronism feeder (SF)*** envelope, while the slave searches for the SF pattern in its incoming data stream. Obviously, its detection soon follows Physical Layer symbol synchronization. The slave then replies by continually sending SF envelopes to the master. When the master detects the SF envelope, it sends another specially coded ***OK*** or ***synchronization acknowledgement*** envelope, whose twofold task is to inform the slave that it has established frame synchronization and to initialize the serial number sequence from master to slave. This is followed by normal information bearing or dummy envelopes. The slave then replies with a corresponding OK envelope, which initializes the serial number sequence from slave to master, followed by normal envelopes. This procedure can also be used if the channel has to be resynchronized after a prolonged noise burst or line break. Both sides retransmit all their unacknowledged packets. Thanks to the sequence number, the receivers are able to pick up the packet stream following the last envelope which had been received correctly before the link had to be resynchronized [NERI77].

2.3.3 Channel Management

The management of the channels forms part of the overall node management and, therefore, it can communicate with other management entities both in the same and remote nodes by means of MCP transactions (Section 2.6). However, it also uses NTAN messages to communicate with channel management entities in adjacent nodes. Since the node management is implemented using a general-purpose microcomputer, the channel management typically operates at a much slower rate than the dedicated high-speed processor, the ***channel controller***, which controls the packet-by-packet protocol of the channel.

The operation of the channel management can be split into two

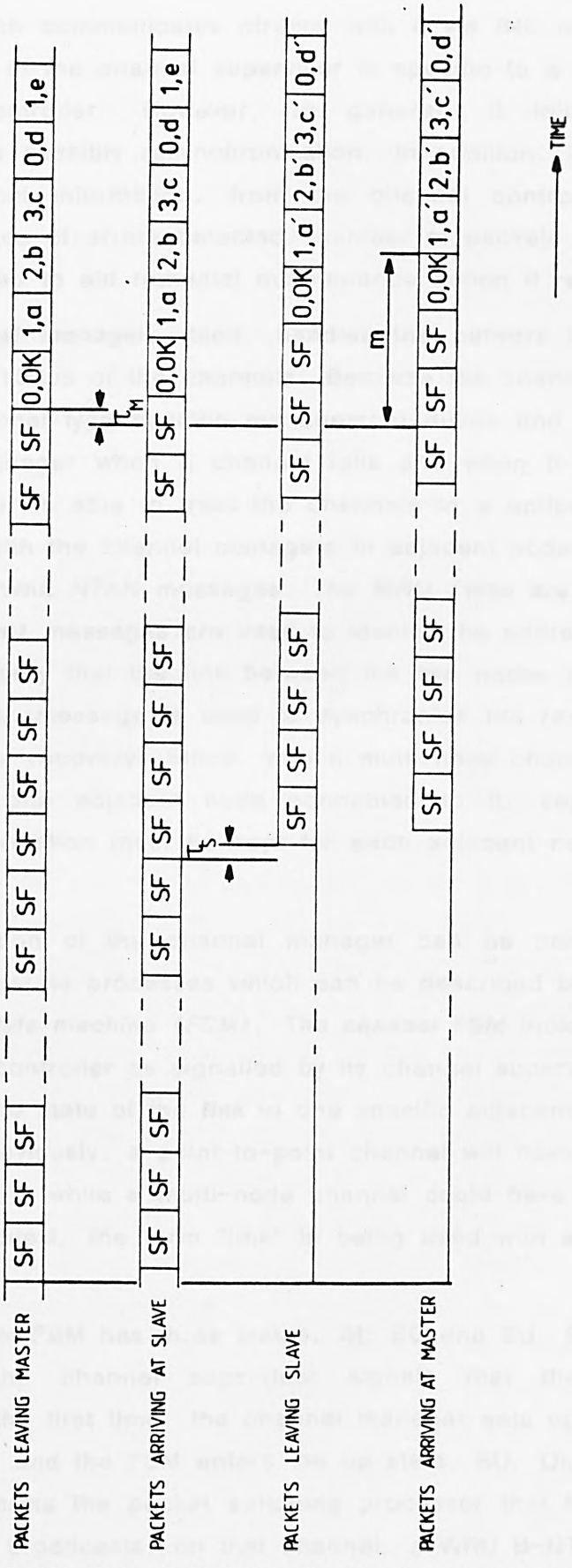


Figure 2.16: Frame Synchronization Procedure (m=2)

separate types of task. Each channel controller has a ***channel supervisor***, which communicates directly with it via INC messages. The exact operation of the channel supervisor is specific to a particular type of channel controller. However, in general, it initiates channel initialization and possibly resynchronization. In addition, it collects and records statistical information, from the channel controller, such as number and types of errors detected, number of packets handled, etc. This can be used to aid remedial maintenance action if required.

The ***channel manager***, itself, handles the network implications of the operational status of the channels. Because the channel supervisors handle the channel-type specific management duties and merely inform the channel manager when a channel fails and when it recovers, the channel manager is able to treat the channels in a uniform manner. It communicates with the channel managers in adjacent nodes by means of WRU, IAM and WKE NTAN messages. The ***WRU (Who are you?)*** and its reply, ***IAM (I am)*** messages are used to identify the address of adjacent nodes and to verify that the link between the two nodes is operational. The ***WKE (wake)*** message is used to synchronize the re-use of a link following channel recovery. Since, with a multi-node channel, there will be more than one adjacent node connected to it, separate routing connectivity information must be kept for each adjacent node connected to that channel.

The operation of the channel manager can be considered as a collection of separate processes which can be described by two different types of *finite state machine (FSM)*. The ***channel FSM*** indicates the state of the channel controller as signalled by its channel supervisor. The ***link FSM*** indicates the state of the ***link*** to one specific adjacent node through that channel. Obviously, a point-to-point channel will have only one link associated with it, while a multi-node channel could have several. Note that, in this context, the term "link" is being used with a very specific meaning.

The channel FSM has three states, SI, SD and SU. SI is the initial state. When the channel supervisor signals that the channel is operational for the first time, the channel manager sets up the database for that channel and the FSM enters the up state, SU. On entering SU, the process informs the packet switching processor that the channel is operational and broadcasts, on that channel, a WRU B-NTAN message, while starting the WRU-IAM timeout for every link through that channel.

On receipt of a WRU message, the manager replies with an IAM S-NTAN message without change of state. When a channel failure is signalled by the channel supervisor, the process informs the packet switching processor and all the link FSMs operating through that channel of the failure, and enters the down state, SD. Periodically, while in SU, the process will repeat the WRU transmission in order to ensure that the failure of an adjacent node (leaving the channel operational) is detected.

The routing management algorithm, described in Chapter 4, operates in terms of links to an adjacent node via a channel. The link process is responsible for informing the routing manager, when the link fails and when it recovers. A problem with all types of network is the marginally operational link, which continually fails and almost immediately recovers. Without safeguards, such a channel could send the network into paroxysms, as it attempts to adapt to the ever changing topology. In order to avoid this, the link process practises a ***link hold-down*** reflex, which delays declaring the link recovered until a time interval has elapsed since the link failed. The length of this hold-down period depends on whether an alternative pathway has been found around the failure. In order to synchronize the re-establishment of a link, the link process transmits a WKE S-NTAN message to its opposite number at the other end of the link, which forces it to also declare the link operational. The link FSM can be simplified to the four-state machine described in Figure 2.17 and Table 2.2. It is driven by failure of the channel, timeouts, reception of IAM and WKE messages and the routing table entry for the linked node. When the channel fails or the maximum number of WRU-IAM timeouts is exceeded, the algorithm immediately sends a ***link failure (lfl)*** message to the routing manager and goes to state SD. Channel recovery does not directly trigger the link recovery process. Instead, the link FSM waits until the WRU, transmitted after channel recovery by the channel process, invokes the IAM reply from the linked node. When this IAM message is received, the algorithm waits in state SH1 until a period TH1 has elapsed since the channel failed. Only if the routing table indicates that no alternative route could be found to the linked node, does it go to state SU, sending a ***link recovery (lrc)*** to the routing manager and a WKE message to the linked node. Otherwise, the algorithm waits for the longer period, TH2, in state SH2, before going to SU and sending the lrc and WKE messages.

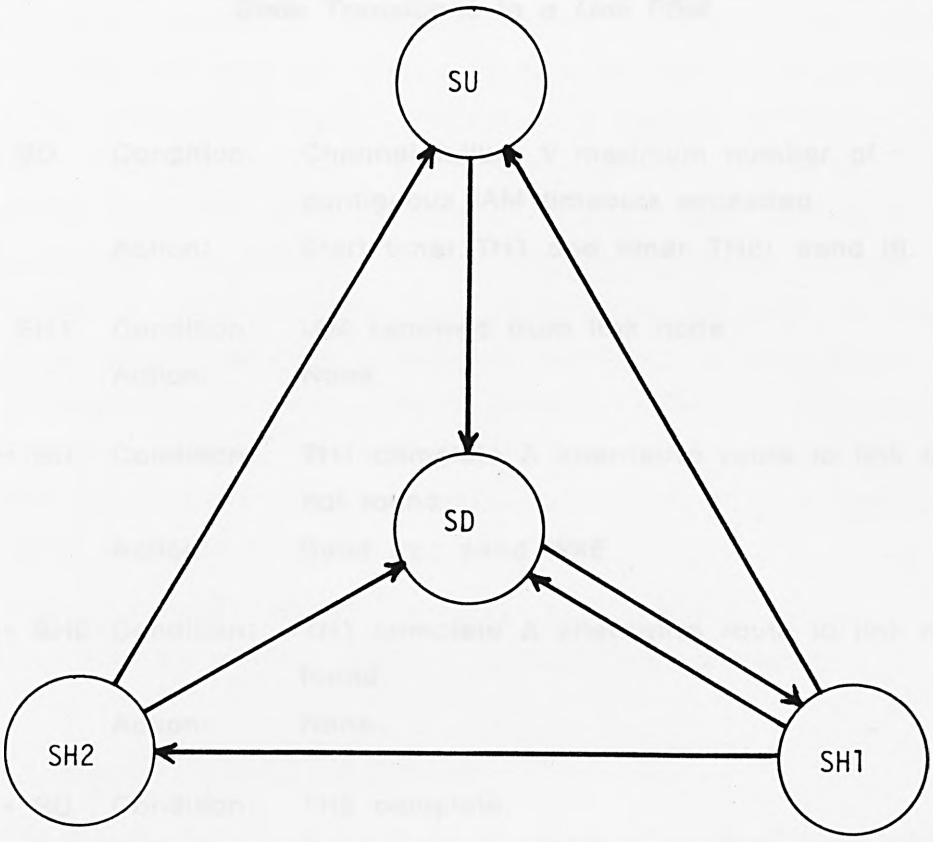


Figure 2.17: State Diagram of the Link Hold-Down Algorithm

The shorter period, TH1, should be roughly equal to the maximum time expected for the routing management algorithm to find an alternative route, if it exists. The longer period, TH2, allows congestion, arising as a result of the original failure, to be dissipated along the new, albeit longer path before triggering another routing change. Receipt of a WKE message causes the process to go immediately to SU sending a lrc message.

2.4 THE PACKET DELIVERY SERVICE

The Packet Delivery Service, provided by the Packet Switching Sublayer, can be summarized as follows:

(1) Its primary function is to transport packets from a source node to any destination node in the network without any loss of sequency.

(2) The delivery delay time is minimized by finding the minimum delay route, between source and destination, and by minimizing the

Table 2.2

State Transitions in a Link FSM

SU → SD	Condition:	Channel failure v maximum number of contiguous IAM timeouts exceeded.
	Action:	Start timer TH1 and timer TH2; send lfl.
SD → SH1	Condition:	IAM received from link node.
	Action:	None.
SH1 → SU	Condition:	TH1 complete Λ alternative route to link node not found.
	Action:	Send lrc; send WKE.
SH1 → SH2	Condition:	TH1 complete Λ alternative route to link node found.
	Action:	None.
SH2 → SU	Condition:	TH2 complete.
	Action:	Send lrc; send WKE.
SH1 → SD	Condition:	Channel failure.
SH2	Action:	Reset and start timer TH1 and timer TH2.
SH1 → SU	Condition:	WKE received.
SH2	Action:	Send lrc.

buffering delay in the relay and end-point nodes.

(3) Overall throughput does not fall significantly as offered load increases to saturate the network. The network is guaranteed free from store-and-forward deadlock by means of the buffer allocation and flow control algorithms.

(4) There is no limit on the number of Virtual Connections routed along any pathway in the network. Consequently, a Network Connection request is never refused on the grounds of buffer non-availability in the Exchanges.

Note, that this service is only concerned with the transportation of packets from source to destination **node**, and does not distinguish between user and network packets or between different destination ports within the destination node. This is an important difference between MININET and many other connection-orientated networks such as TRANSPAC [DANE76], GMDNET [RAUB76] and TYMNET [TYME71] where, on connection establishment, state information concerning the Virtual Connection is stored in every relay node along the path. If this latter method is used, then: the connection procedure is complicated by the direct involvement of relay nodes; there is a danger that connection establishment can be refused if an intermediate node has already allocated all its buffers; and connections are lost as a consequence of relay node or channel failure.

The congestion control strategy is described in the following section, while the routing algorithm is described in Chapter 4.

2.4.1 Congestion and Flow Control

Congestion is here defined as an overload of a network resource, of which there are three major types: node processing power, buffer storage space and channel bandwidth [MCQU79], [NESS79]. It is the object of the congestion control mechanisms to handle these overloads in such a way that there is no reduction in total packet throughput, no discarding of packets and no loss of fairness or sequency (Sections 1.2.2 and 1.2.3). While the routing strategy does have an indirect effect on congestion by maintaining the shortest, loop-free path between source and destination, it is not used to **control** congestion within MININET. The primary control mechanism is flow control.

Two flow control mechanisms are used in MININET. One, the ***channel flow control***, is provided by the Channel Service to match the effective channel transmission rate to the speed at which the receiving node can process the incoming packets. Thus, processor overload is avoided. If this were the only flow control mechanism provided within the network, it would be prone to store-and-forward deadlock [KAHN72]. For this reason, the ***network layer flow control*** is provided. Its objective is to avoid buffer overflow within the Exchanges, while keeping the paths between source and destination nodes deadlock free. In order to achieve this, some sort of buffer partitioning, within each Exchange, must be

used [MERL80] [GERL80].

In ARPANET, *direct store-and-forward deadlock* between two adjacent nodes is avoided by reserving one buffer within the switch for each output channel, and two buffers for each input channel. This reduces, but does not remove, the possibility of *indirect store-and-forward deadlock* involving more than two nodes [GERL80]. Two additional buffers are employed to hold "overflow packets", which can be used to attempt to break the log-jam should deadlock occur. As a final resort, a reset, with consequent loss of packets, occurs [KAHN72]. In CIGALE, which is the packet switching network of the CYCLADES computer network [POUZ74], the length of each output queue (one per output channel) is restricted. This does not guarantee freedom from lockup. Instead, each packet is "time-bombed" and old packets are discarded, thus dissipating the congestion [GRAN79]. One method, which guarantees deadlock free operation, is to reserve at least one buffer at each intermediate node when a Network Connection is established, as in TYMNET [RIND79]. Unfortunately, this has the disadvantage of under-utilization of buffers.

Perhaps the first published attempt to devise a more efficient buffer allocation scheme, that is guaranteed deadlock free, was the *buffer class* method [GUNT75]. This was used in GMDNET [RAUB76]. For each hop undertaken by a packet, its "class" increases by one. Buffers are reserved in a node for each class. A packet is permitted to use buffers within its own class and any lower class. A modification of this method is to base the buffer class on how many hops the packet has to go to its destination, rather than on how many it has traversed from its source [TOUE79]. Merlin and Schweitzer [MERL80] generalized these methods as special cases of the *buffer graph* approach which, by means of a number of possible methods, seeks to arrange the buffers within the network into a directed graph. Consequently, there are no directed loops and there exists at least one directed path corresponding to each route in the network. Obviously such a graph is deadlock free. In the proposed MININET flow control algorithm, buffers are reserved on a destination node basis. Thus, each exchange contains up to 63 buffer partitions, each one containing only packets destined for one particular node. Since these packets will occupy only buffers associated with that destination, the network buffer graph decomposes into up to 63 disjoint graphs - one per destination node. Provided that the routing algorithm

is loop free (which it is), each of these graphs forms a directed *tree* rooted at the destination. Consequently, the network is guaranteed deadlock free. The destination-based buffer class method could also be used. However, the high-speed control of the buffers would be greatly complicated by the mapping between the buffer class and BPVs. Within each buffer partition, the waiting packets form the ***destination queue.***

Given some buffer partitioning scheme, there remains the need for a flow control method to avoid overloading the buffers within each partition. Global anti-congestion approaches, such as the *isarithmic flow control* method [PRIC77] which seek to limit the **total** number of packets in the network, are discounted because there is no guarantee that local congestion cannot occur. In connectionless networks such as CIGALE, there is a tendency to discard packets if the congestion gets bad. CIGALE attempts to avoid reaching that stage of congestion, by sending *choke packets* back to the source hosts requesting them to reduce their rate of transmission. Packets arriving in an ARPANET node are examined to see if they can join an output queue. If this is not possible, either because there is a shortage of buffers within the node, or because the maximum number of packets associated with the output channel would be exceeded, the packet is refused. This refusal is implemented by the absence of an acknowledgement of the packet to the upstream node. Subsequently, the upstream node retains its copy of the packet and retransmits it after a timeout. Note that this type of flow control depends upon a **Network Layer** hop-by-hop packet acknowledgement protocol because, as far as the Data Link Layer is concerned, the packet was received correctly and passed to a Network Layer buffer for examination within the node core.

In TYMNET, each node is not allowed to transmit more than a certain number of characters belonging to a Virtual Connection, without receiving an indication from the next node that buffer space is available. This indication is made in the form of a *backpressure vector* containing a single bit for each connection. If there is room in the connection's queue within the downstream node, the bit is set to one. Reception of this, by the upstream node, allows it to transmit characters, up to the limit set for that connection. If the connection's queue in the downstream node is over this limit, the corresponding bit of the BPV is reset. In the upstream node, this will lead to the halting of transmission for that connection. Eventually, the connection's queue within that node

will exceed its limit, thereby triggering the application of backpressure to its upstream node. Thus, backpressure propagates backwards along the path of the Virtual Connection until, if necessary, it reaches the source. Note that buffer space, up to twice the backpressure limit, must be reserved for the connection in each node, and that the product of this limit and the rate of backpressure vector production determines the maximum throughput for that connection. While this **passive** backpressure vector approach is admirably suited for the relatively low-speed terminal traffic handled by TYMNET, the high burst rates of some instrumentation traffic make it unsuitable for use in MININET. If the worst case example of 100k packets per second (Section 1.2.3) is taken as the peak Virtual Connection rate and, say, an absolute maximum vector update rate of 2k per second (1000 times faster than TYMNET) is allowed, then 100 buffers would have to be reserved for each connection in each relay node. Instead, a destination node based **active *flow vector*** is used. This is transmitted only and immediately when its contents are changed.

As described above, each MININET Exchange contains a separate output queue for each destination in the network (up to a maximum of 63). Disjoint subsets of these queues are attached to the output channels, according to the routing algorithm as shown in Figure 2.18. An output switching process polls the queues associated with one output channel on a round robin basis. The input switching process places incoming packets at the back of the appropriate queue. The Exchange maintains its own flow vector, which is a function of the destination queue lengths. If the input switch detects that a queue has reached its upper limit, the corresponding bit of the flow vector is cleared. Similarly, if an output process detects that a queue has reached its lower limit, the corresponding bit is set. Any change to the flow vector triggers its transmission to all adjacent nodes using BPV B-NTAN messages (Figure 2.8). Bit 0 of BPV code 0 pertains to node 0. Bit 15 of BPV code 0 pertains to node 15 and so on, until bit 15 of BPV code 3 which pertains to node 63. As each BPV message holds only one quarter of the complete flow vector, only those 16-bit segments of the flow vector that have changed need be transmitted. Received BPVs are used by the output switch corresponding to the channel from which the BPV was received. Section 2.3.1 describes the special processing of BPVs within the channel controllers of multi-node channels. A packet

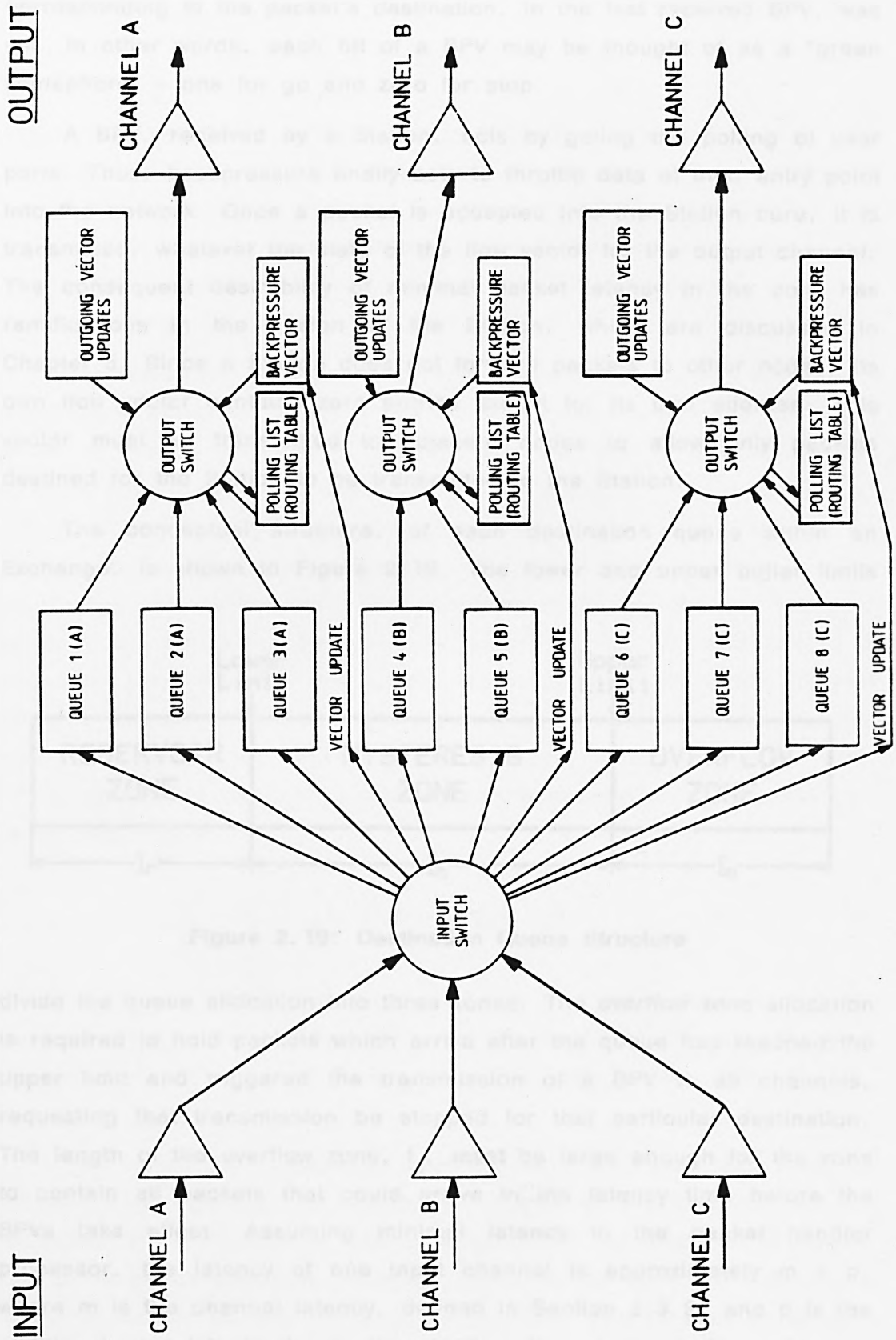


Figure 2.18: Exchange Buffer Structure

may be transmitted to an adjacent node provided that the bit, corresponding to the packet's destination, in the last received BPV, was set. In other words, each bit of a BPV may be thought of as a "green semaphore" – one for go and zero for stop.

A BPV, received by a Station, acts by gating the polling of user ports. Thus, backpressure finally acts to throttle data at their entry point into the network. Once a packet is accepted into the Station core, it is transmitted, whatever the state of the flow vector for the output channel. The consequent desirability of minimal packet latency in the core has ramifications in the design of the Station, which are discussed in Chapter 5. Since a Station does not forward packets to other nodes, its own flow vector contains zero entries except for its own address. This vector must be transmitted to adjacent nodes to allow only packets destined for the Station to be transmitted to the Station.

The conceptual structure, of each destination queue within an Exchange, is shown in Figure 2.19. The lower and upper buffer limits

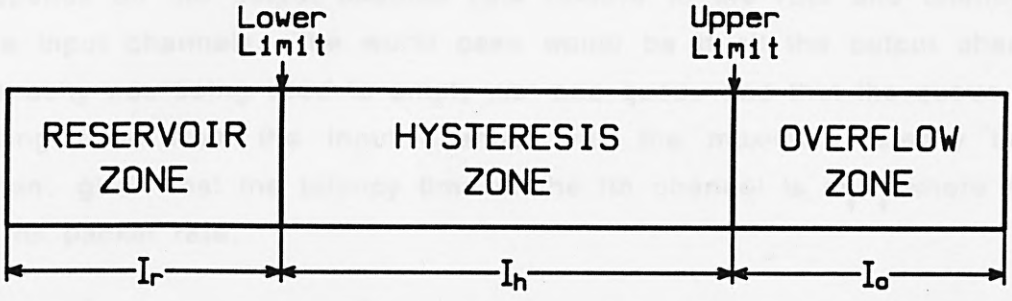


Figure 2.19: Destination Queue Structure

divide the queue allocation into three zones. The ***overflow zone*** allocation is required to hold packets which arrive after the queue has reached the upper limit and triggered the transmission of a BPV to all channels, requesting that transmission be stopped for that particular destination. The length of the overflow zone, I_o, **must** be large enough for the zone to contain all packets that could arrive in the latency time before the BPVs take effect. Assuming minimal latency in the packet handler processor, the latency of one input channel is approximately m + p, where m is the channel latency, defined in Section 2.3.2, and p is the additional ***core latency*** due to the pipeline delay between the node core and the channel controllers. The minimum possible value of p is 2. It could be higher due, most likely, to transmission latency arising from

delays in distributing outgoing BPVs to all channel controllers. Therefore,

$$l_o = \sum (m_i + p) \qquad (2:11)$$

where the summation is taken over all input channels and m_i is the latency of the ith channel. Note that, as far as buffer allocation is concerned, half-duplex channels (m=0) are preferable because of their low latency. If it is conservatively assumed that the average channel latency is 2, that the core latency is 3 and that the maximum number of input channels is 8, then, from (2:11), $l_o = 40$.

The ***reservoir zone*** is required to allow the output channel to continue to empty the queue in the latency period following the issuing of BPVs to restart the flow of packets into the queue, but before new packets actually arrive. If the length of this zone, l_r, is insufficient, the queue will be exhausted before the new packets arrive. This has the effect of reducing the throughput of the output channel, further increasing the congestion. The required minimum size of this zone depends on the output channel rate relative to the rate and latency of the input channels. The worst case would be if all the output channel capacity was being used to empty just one queue and that the queue was being filled from the input channel with the maximum latency time. Then, given that the latency **time** of the *i*th channel is m_i/f_i where f_i is its packet rate,

$$l_r = \left\lceil f_o \, \mathrm{Sup} \, (m_i/f_i) \right\rceil + p \qquad (2:12)$$

where f_o is the output channel packet rate and the supremum is taken over all input channels. Assuming that the maximum channel latency is 3 and that all channels have the same rate gives, from (2:12), $l_r = 6$.

The larger the ***hysteresis zone*** the less frequently will BPVs be issued. Since channel throughput is reduced by the overhead of BPV transmission, throughput is maximized by having the length of this zone, l_h, as large as possible. However, in order to minimize end-to-end delay, l_h should be kept as small as possible. This is another example of the conflict between throughput and delay, discussed in Section 1.2.3. The maximum BPV rate for a given value of l_h occurs

when the queue is being filled and emptied at maximum rate with minimum latency. Analysis of these worst case conditions gives an estimate for the maximum required length of the hysteresis zone. Minimum latency is obtained with m = 0 (half-duplex channels) and p = 2. Worst case conditions can be assumed to be an Exchange with 8 half-duplex channels, all operating at 100k packets per second. If it is assumed that the queue is being continuously emptied by one channel, while being filled, backpressure permitting, by the other 7 channels, then the rates at which the queue is filled and emptied are 100k and 600k packets per second respectively. The over-run into the reservoir and overflow zones are 2 and 14 respectively. Thus, the queue length would vary by $I_h + 14 + 2$ packets. Hence, the queue filling time is $(I_h + 16)/600$ ms and the emptying time is $(I_h + 16)/100$ ms. In one fill-emptying cycle, two BPVs are transmitted. Therefore, the per channel BPV transmission rate in kilo-messages per second, B, is given by

$$B = 2\left[\frac{I_h + 16}{600} + \frac{I_h + 16}{100}\right]^{-1} \simeq \frac{171}{I_h + 16} \qquad (2:13)$$

Rearranging (2:13) and rounding up,

$$I_h = \left\lceil \frac{171}{B} \right\rceil - 16 \qquad (2:14)$$

Equation (2:14) indicates that, assuming a maximum allowable BPV transmission rate of 2k per second (2% of channel throughput), the maximum required size of the hysteresis zone is 70.

The total number of packet buffers, that might be required by the queue, can be obtained by adding the maximum lengths of the three zones. For the (slightly contradictory) worst-case conditions outlined above, this gives a total queue allocation of 116 buffers. However, there can be problems with this type of active flow vector if an Exchange has channels with large differences in capacity. Since the BPVs are distributed to all channels, a high rate of BPV production, associated with a high-speed channel, could well take up a considerable part of the capacity of a low-speed channel. If there is a danger of this, a much larger hysteresis zone would be required. Note that, if a low-speed channel is the cause of the congestion, there is no problem because the queue emptying rate is low and consequently the BPV generation rate

is low.

Buffers can be allocated to a destination queue, and the upper and lower trigger points can be set, either statically or dynamically, in a number of ways. In selecting a method for implementation, consideration must be given, not only to the efficiency of buffer utilization, but also to the complexity of the scheme. This is because a specially designed switching processor would be required, in order to achieve the desired Exchange packet switching capability which is well in excess of 100k packets per second. Some options are:

(a) The parameters could be fixed in the design of the Exchange. Since the number and type of channels are unknown at design time, some worst-case assumptions have to be made, as outlined above, in order to determine the sizes of the three zones. Furthermore, since the network configuration is also unknown, a separate allocation must be made for all (63) potential destinations. Obviously, this results in very poor buffer utilization, but the input and output switch algorithms are the simplest possible. That the buffer under-utilization need not be very important, can be appreciated by considering a possible implementation where the buffer space is organized as an 8K x 32-bit memory. This can be implemented with only 4 chips and can contain 8K packets. The memory could be partitioned into 64 128-packet allocations, one for each destination address. This allocation is 12 more than that required for the worse-case conditions described above. Mapping from destination address to queue location is very simple and there is no dynamic queue allocation process to run.

(b) During initialization, information on the number of channels and their speed and latency can be obtained from the channel manager. The optimum values of I_r and I_o can then be calculated using (2:11) and (2:12). This allows I_h to be maximized. The total queue allocation remains fixed as in option (a).

(c) Some form of dynamic allocation can be used. Quiescent destination queues are given the minimum safe allocation, which is $I_o + 1$. A number of buffers are held in reserve. When a hitherto quiescent queue begins to fill, instead of immediately applying backpressure, buffers are allocated to the newly active queue up to

some predefined limit, assuming that the contingency reserve has not been exhausted. Further traffic-based allocations can be made and buffers removed from newly quiescent queues, by a buffer management entity operating much more slowly than the high-speed packet handling processor. The dynamic nature of the buffer allocation means that the queues would have to be implemented as linked lists, instead of the simpler contiguous structures which can be used with static partitioning.

Option (c) would theoretically utilize the buffer memory more efficiently, but at the unacceptably high price of much greater processing complexity. Therefore, dynamic allocation is not recommended. On the other hand, option (b) provides some improvement over the performance of option (a), with little increase in complexity. Therefore, option (b) is recommended for this application.

The simplest and most obvious internal organization of the destination queues is a standard FIFO queue. However, with this type of structure, burst mode traffic can fill the queue and so significantly delay handshake traffic. The throughput of handshake traffic is most sensitive to delay (Section 1.2.3). One method of reducing the interference of burst mode traffic on handshake delays is to use a two-dimensional queue. This consists of a separate FIFO queue for each destination **port** within the destination node, plus one for network packets. These are linked together at the front end to form a loop of queues. The output switching process uses this loop to remove packets from the queue on a round robin basis. The effect of this queue structure is to give top-priority to the first packet of each Virtual Connection in the queue. Consequently, sequency would no longer be kept in the packet flow between nodes, although the sequentiality of the Network Connection is preserved. As far as queue size and buffer allocation are concerned, the destination queue is treated as a single entity. The primary disadvantage of this approach is complexity. Each packet buffer would have to contain two link pointers. The input switch would have to examine the destination port field, in addition to the destination node field, of each incoming packet, and would have to maintain pointers to the back of 65 different queues for each destination node.

End-to-end flow control, in the sense of source-sink data rate matching, is not directly provided by MININET. To do so would

compromise the transparency requirement (Section 1.2.2) because, when a pair of user devices are directly connected together, they must be able operate their own flow control procedures without the aid of the network.

2.5 THE MININET SERVICE

The MININET Service, provided by the Virtual Connection Sublayer, can be summarized as follows:

(1) Its primary function is to maintain a transparent Virtual Connection between two devices, which are physically connected to two ports of the network. The degree of transparency is such that the devices are not necessarily aware of the network's presence and communicate as if they were directly connected together.

(2) Virtual Connections are established and closed by network management, at the request of a user management entity. This request is actually made either via the operator console or via a network management port, neither of which need be connected to either of the end-point nodes.

(3) Failure of an intermediate node or channel does not cause the loss of the Virtual Connection, provided that an alternative pathway exists between the end-point nodes, although some data loss could take place.

2.5.1 Virtual Connection Management

The di-phasic connection establishment procedure, used in MININET, has already been described in Section 2.1.3. Connection service interactions, between the user and network managements, have been implemented in two ways. One type of MSAP is effectively the node operator console, where the operator represents user management. The connect request primitive is assembled interactively by the operator and the connect confirmation is displayed at the same console. It is not necessary for the console to be situated at an end-point node. The other type of MSAP is a ***management port.*** This is a port which has been virtually connected to its local node manager. Thus, a host computer, connected to a management port, can request connection changes associated with any pair of ports in the network. This enables extra

higher-level services, such as host switching for interactive terminals and dynamic resource sharing between processors, to be implemented.

A port may be in a number of states, as far as the status of any Virtual Connection made to it is concerned. It may be ***inactive***, i.e. not connected to anything. It may be connected to another port which could be located in the same node, in which case it is referred to as a ***local connection***, or in some other node, in which case it is called a ***remote connection***. It may be a management port connected to the node manager. This allows user management to communicate with network management and thus make changes to Virtual Connections, obtain network statistics, etc. The ***standby*** state is used to enable additional services to be provided by higher level entities. The port is connected to the node manager in just the same way as a management port. However, the content of any data sent to the manager is ignored. Instead, its reception triggers the manager to connect the port to a destination, whose address has already been stored in the manager. In contrast to the strict one-to-one rule on active connections, any number of ports over the network can be on standby to the same destination. An example of this mode of operation is the provision of a terminal server. All inactive terminals are on standby to a port connected to the terminal server. When any one of these terminals becomes active (i.e. by its user pressing any key), it is automatically connected to the terminal server, provided that the latter is not already connected to some other terminal. By means of interactive messages, transparent to the network, the terminal user can request the server to connect the terminal to its desired host computer. This is done by means of a separate management connection to the terminal server.

Each node stores state information, concerning the Virtual Connections terminating at that node, in its ***virtual connection table (VCT)***. There is an entry in this table for each port physically connected to the node. This contains:

(a) The connection status of the port. This may be inactive, remote, local, management, standby or suspended. The latter state is entered when the routing protocol determines that the destination node is not currently reachable in the network. If the VCT is in a Station, the remote state is further sub-divided according to the output channel used by each Virtual Connection, as determined by the routing protocol.

(b) The destination node and port address.

(c) The attributes of the connection. If the protocol used by the connected devices conforms to DIM-CPC, the network management can report certain exception conditions directly to either one of the connected devices, following the format described in Section 3.3.3. In the typical case of the connected devices being a computer and a peripheral, the error messages are usually sent to the computer. The ***report local*** flag, within the port's attributes, indicates that the error messages should be sent to the port itself, while the ***report remote*** flag indicates that any error messages should be sent to the port at the other end of the connection.

(d) A password. This is a short password, which must be correctly quoted by any user management entity requesting the modification of the connection status of the port. It is intended to provide protection against the user accidentally specifying the wrong port, rather than any **deliberate** misuse of the network.

Additional entries in the VCT, used internally by the node management, are concerned with the implementation of the connection information in the high-speed packet handling hardware (e.g. location in polling lists).

When a connection request (or disconnection request) is made via the node's operating console, the request is first assembled interactively with the operator. This is then dispatched to the ***VCT manager*** in the node first referenced in the request. Note that this may, or may not, be in the same node as that originating the request. If it is not, MCP (Section 2.6) is used to transport the request to the node. The VCT manager first checks that the request is acceptable as far as it is concerned. If the connection request is for a remote connection, it then sends a connect message, using MCP, to the VCT manager in the second node referenced. The second VCT manager replies indicating whether, or not, the request is acceptable and has been implemented. On receipt of a positive reply, the first VCT manager implements the change. In any case, the success, or otherwise, of the request is relayed back to the originating task and hence to the user management. Had the request concerned a local, management or standby connection, the VCT manager can implement the entire request without recourse to any other manager. Connection requests, arising from a user machine

communicating via a management port, are handled in an identical manner.

2.5.2 Network Ports

The main objective of the MININET Service is that the network should be as transparent as possible so that it is procedurally invisible to the devices communicating through it. In principle, any type of interface can be used, provided that the interface protocol does not demand the impossible from the network. In particular, any "read" type operation, where one side requires information from the other side within a very short time (typically considerably less than 1μs) of activating a control strobe, cannot be implemented across any network. This is because the requested information cannot be transported across the network within the time allowed.

A simple, bidirectional port status bus, quite separate to the packet data transfer bus, allows communication between the ports and the node manager. This is used by the ports to signal device timeouts and interface transmission (parity) errors to the manager, while the manager can acknowledge these and selectively reset any port.

The DIM interface has been specially designed so that a network could be interposed between the two sides of the interface without any change to the interface protocol (Section 3.1). Therefore, the design of the DIM port is quite straightforward, without any protocol explicitly operating between the ports themselves. Instead, information arriving along the 16-bit data lines and the data/control shift line is mapped straight into the data field and data class flag of a packet. The only actions performed by the DIM port are: to buffer incoming and outgoing words, to detect when a user device fails to respond, and to detect parity errors across the interface and port bus.

The IEC-625 (IEEE-488) instrumentation bus [IEC 79] is a single source (the *talker*), multiple destination (the *listeners*), byte-serial bus. The talker and listeners are selected by the bus *controller-in-charge* which, by asserting the "attention" *ATN* control line, becomes the source, and broadcasts control set-up information to all devices. Data transfers are synchronized by means of 3 handshake lines: *DAV* controlled by the *source* (the talker or controller), and *NRFD* and *NDAC* controlled by the *acceptors* (only the listeners during normal data

transfer - all devices when the controller is active). The bus was not designed to be used across a network. Therefore, an ***IEC-625 port*** must use an additional protocol in order to obtain procedural invisibility. The port always acts as an acceptor. This enables it to capture any data transfers on one side of the bifurcated bus and relay them to the section of bus on the other side of the network, where the port acts as a source. The inter-port protocol uses handshake mode (Section 3.3.1) to interlock the flow of data between the ports, so that not more than one byte of information is buffered within the network or ports at any one time. This is linked with the handshake control lines of the bus, thus completely interlocking the flow of information along the bus. Consequently, the maximum information rate along the bus is restricted to the reciprocal of twice the end-to-end network delay. The information flows from the talker in normal mode and from the controller-in-charge when it asserts ATN. The value of ATN is relayed by the ports from the controller side of the network to the other half of the bus. Similarly, the value of the service request *SRQ* line is relayed to the controller side by the ports. Note that the controller and talker may well be on different sides of the network, implying that data could flow in different directions depending on the type of transfer.

The bus *system controller* controls the interface clear (*IFC*) and remote enable (*REN*) bus control lines. These are relayed from one side of the bus to the other by the ports. The location of the system controller, and hence the direction of transfer of these signals as well as the initial direction of ATN and SRQ, is automatically determined when the system controller initializes the bus by asserting IFC. Each data packet travelling between the ports carries, in addition to the 8-bit bus data byte, two flags. One indicates the value of the *EOI* line which is used to flag the end of a block transmission. The other is used to set the ATN line on the destination port, when the data byte is a controller command. The parallel poll function cannot be implemented because it requires a solicited response within 200ns of the request being issued (via the EOI line). Clearly, this requirement cannot be physically met when the information required lies some hundreds of metres away on the other side of the network.

Packetized speech can be carried through the network using the ***speech port.*** Two of these can be virtually connected through the network to be used as an full-duplex intercom. Also, it is possible to

connect virtually a speech port to a computer, via a DIM port, to allow recorded messages to be broadcast. The speech signal is sampled at 8kHz, using a standard PCM CODEC, to form an 8-bit companded sample. Two of these samples are concatenated into a 16-bit data word which fills the data field of the packet. Thus, the peak load on the network is 4k packets per second. However, data is not transmitted continuously at this rate because the transmission is voice keyed, and so the network is loaded only when the input signal level becomes greater than a preset threshold value. The data is transferred between the speech ports in burst mode (Section 3.3.1), and a FIFO is used by the receiver to buffer short term fluctuations in the network transit delay. The CODECs are clocked independently by separate crystal oscillators. Therefore, there must be some slight difference in operating frequency between any two ports. This rate mismatch is handled by the occasional loss or duplication of a sample. This does not seem to be noticed by the users. When virtually connected to a computer, the data transfer would be synchronized by using handshake mode flow control.

2.6 THE MANAGEMENT TRANSPORT SERVICE

Each node manager consists of a number of distinct ***tasks***, each concerned with separate jobs such as: supervising the channel controllers and packet handling hardware; maintaining the routing table and VCT; handling the operator console interactions; and so forth. The implementation of the Station manager is described more fully in Section 5.3. Inter-task communication and task synchronization are effected by the exchange of various types of message. The ***Management Transport Service*** effectively extends the scope of one type of inter-task message to the entire network. This service is transparent to the extent that the tasks are unaware of any procedural differences in the transfer of messages between tasks whether they are in the same node or in different nodes. An end-to-end protocol, the ***MININET control protocol (MCP)***, [MORL79] is used to provide this Transport Layer Service.

In the context of this protocol, a ***network message*** is defined as a block of information carried from one node manager to another by a series of network packets (Section 2.2.1). A ***network conversation*** is a series of network messages sent **alternately** between two conversing nodes to effect one management operation. The originator of a conversation is called the ***caller***, while the other party involved is

designated the ***callee***.

This service and protocol have the following constraints and requirements:

(1) Since a network conversation is invariably of the "question-and-answer" or "request-action" type, there is no point in providing a full-duplex message service. Instead, the messages should be exchanged alternately.

(2) The network packets, used by MCP, must be the same size and use the same Packet Delivery Service as the user packets. The consequent small size of this PDU means that the SDU must be segmented and reassembled at the other end.

(3) MCP must not significantly interfere with normal traffic, i.e. network packets must not hog or block any network resource.

(4) It must be able to recover and continue after damage to, or loss of, a network packet.

(5) Total failure of the caller or callee, or total loss of communication between them (due to channel or relay node failure), must not cause the protocol handler or the communicating tasks to be indefinately suspended.

(6) Since the integrity of the whole network operation depends on the reliability of MCP, because of the sensitivity of the information it carries, end-to-end error protection at a message level is desirable.

(7) Since the very time-critical operations of routing and flow control do not use MCP, speed is not as important requirement as that of simplicity.

Because the Management Transport SDUs are constrained to operate alternately in a half-duplex fashion, the service provided is not a conventional connection type. In fact, it can be thought of as an extension to the *user-confirmed* connectionless type of service [VISS85], which restricts the number of SDUs to a single message from the caller and a single reply message from the callee. There is no such restriction with this service. However, in practice, no application has used more than two messages of the request and answer type. An originating task passes its message to a local distribution routine which, either sends

the message directly to the destination task, if it is in the same node, or to the transport server, if it is not. Only two service primitives are required: data request and data indication. These consist of: address fields specifying the source and destination nodes and tasks; flags to indicate that a reply is expected and to indicate whether failure to transport the message should be reported to the source task (used by the data-request primitive only); status of message, indicating if there was an unrecoverable error while attempting to transmit the message or receive its reply (used by the data-indication primitive only); length of the message in 16-bit words; and the (variable-length) message itself.

In order to keep the transport server simple, a node can be involved in only one conversation at a time. If the node is not engaged in a conversation, the Transport server task is in ***broadcast mode*** and it is open for the reception of a message, either as a service request from another task in the same node in which case it becomes the caller, or a new incoming message from any other node in which case it becomes the callee. The reception of either of these places the server into ***privileged mode***. It then responds only to network packets from the other node in the conversation, and queues any other requests to open a conversation. If a data request primitive indicates that a reply is expected, the server stays in privileged mode and awaits a reply.

In order to avoid flooding the network with a block of network packets forming a message, and to enhance the reliability of the service, network packets (as well as messages) are transmitted on a half-duplex basis. The phase bit in each network packet (Figure 2.7) is used as a modulo-2 sequence number by the protocol to implement the NPL [SCAN69] acknowledgement technique. Since this corresponds to MLP (Section 2.3.2) with the channel latency, m, equal to zero, it has the intrinsic simplicity and robustness of that protocol. However, in MCP, there is a possibility that a network packet might be lost (i.e. never delivered to the destination node) somewhere in the network. To overcome this problem, the caller maintains a timer and eventually retransmits its last packet just as if it had received a damaged packet. The period of this timeout must be greater than twice the maximum end-to-end network delay (even following a node or channel failure), plus the maximum latency period in the corresponding node manager. If this were not the case, the original packet could eventually arrive after a retransmission request had already been made for a duplicate. This

duplication would continue, for the duration of the conversation, with every packet being duplicated and two packets being in the network at the same time. If necessary, the caller repeats the timeout retransmissions a number of times, before it decides that the pathway between the two nodes has permanently failed and aborts the conversation. Note, that the callee must not also retransmit after a timeout, since this would result in error message packets being transmitted in both directions at the same time, leading to the same situation as described above. Instead, the callee maintains a timeout, which is longer than the maximum number of consecutive retransmission timeouts performed by the caller.

In order to initiate a conversation, the caller sends a ***hello packet***, which acts as conversation request and places itself into privileged mode. If the callee is in broadcast mode, it accepts the hello packet and replies with a ***hello acknowlegement packet.*** It then switches to privileged mode to block hello packets from other nodes. However, if the callee is in privileged mode, it ignores all network packets not originating from the node with which it is conversing. This can be done because each network packet contains the address of its source node (Figure 2.7). Therefore, the hello packet is ignored and the caller must retransmit the hello packet after an appropriate timeout interval. In the event of two nodes happening to send hello packets at the same time, the packets will be ignored, because each node would have switched into privileged mode, upon hello transmission, and would be expecting a hello acknowledgement. There is a danger that subsequent retransmissions will also be rejected if the hello retransmission is the same for both nodes. In order to overcome this problem, the hello retransmission timeout period is made different for each node by the simple expedient of making them proportional to the node address. The values of the phase bits in the hello and hello acknowledgement packets initialize the expected sequence numbers in both directions.

Following reception of the hello acknowledgement packet, the caller transmits the first packet of its message. The message consists of a two-packet header, a data field containing the SDU and a single packet checksum. The contents of the header are obtained from the data request SDU and consist of the source and destination task identifiers (the source and destination node addresses are already carried in every network packet), a set of flags as specified in the data request primitive

and the length of the message. Each packet of the header and data fields is acknowledged by the callee with a ***message continue packet***, which indicates that the next packet of the message should be sent. Following reception of the packet containing the checksum, the callee responds with a ***message acknowledgement packet***, if the checksum is correct. If this is not the case, a ***message repeat packet*** is transmitted instead. If the checksum repeatedly fails more than a certain number of times, the conversation is aborted by means of a ***message failure packet***.

If a reply is expected, the caller responds to the message acknowledgement packet with a message continue packet. When the callee server receives this and the reply SDU from the original destination task, it starts to transmit the reply message following the same procedure, as already described, with the role of the caller and callee reversed. This alternate message exchange can continue as long as is required by the users of the Transport Service. When, eventually, a reply is not expected, the conversation is terminated. This is not as simple as it may appear in that it breaks the packet interlock and there is no way of acknowledging the last packet. In order to ensure that the last packets of significance are safely received, an additional ***goodbye packet*** is transmitted to signal termination of the conversation and to acknowledge the message acknowledgement packet. Following transmission of a goodbye packet, the server returns to broadcast mode. If the last message is sent by the caller and it does not receive a message acknowledgement packet, it will timeout and retransmit in the usual way. On the other hand, if the goodbye packet, transmitted by the caller following reception of a message acknowledgement packet, is lost or damaged, it is never retransmitted. It is important only because it informs the callee that its message acknowledgement has been received and it can return to broadcast mode. Its loss merely delays, until its timeout interval expires, the callee's return to broadcast mode. If the last message is from the callee, the goodbye packet from the callee, acts to stop the caller retransmitting the last message acknowledgement. If it is lost, the caller will eventually reach the maximum number of retransmissions and return to broadcast mode.

End-to-end error protection is provided by the parity bits in each packet. For the messages, these are reinforced by the checksum. In combination, these longitudinal and transverse checks provide a

Hamming distance of 4. The hello, message continue, message acknowledgement, message repeat, message failure and goodbye packets do not form part of a message and do not, therefore, get additional protection from any checksum. Therefore, their codes are spaced at a Hamming distance of at least 8 from each other. Such a code can be constructed as follows. Since there are 6 distinct codewords to be generated, an information field of, at least, 3-bits is required. Consider a 15-bit binary cyclic code. The *Bose-Chaudhuri-Hocquenghem (BCH) bound* guarantees that the code will have a minimum distance of (at least) 8, if the roots of the generator polynomial include α^0, α^1, α^2, α^3, α^4, α^5 and α^6, where α is a primitive element of the Galois Field of order 2^4 ($GF(2^4)$) [PETE72]. This can be achieved by forming the generator polynomial,

$$g(X) = m_0(X) . m_1(X) . m_3(X) . m_5(X) \qquad (2:15)$$

where $m_i(X)$ is the minimum polynomial containing α^i. Since, $m_0(X)$ is of degree 1, $m_1(X)$ and $m_3(X)$ are of degree 4, and $m_5(X)$ is of degree 2, the generator polynomial is of degree 11. Since the number of check digits is the same as the degree of the check polynomial, the information field length is 4, which is one more than that required. For example, using the tables of irreducible polynomials over GF(2), found in [PETE72], and taking the primitive polynomial, $X^4 + X + 1$, as $m_1(X)$ results in

$$g(X) = X^{11} + X^{10} + X^9 + X^8 + X^6 + X^4 + X^3 + 1 \qquad (2:16)$$

Note, that only 14 bits are required for this code leaving two bits unused. However, attempting to incorporate these bits into a linear code with a greater Hamming distance results in a check field which is too large for this application. Of course, the codeword need not be computed each time it is used because, with such a small code set, it is much easier to use a table look-up method both for encoding and for checking.

Chapter 3

THE DIM INTERFACE

The advantages of an intermediate interface standard for the interconnection of computers and peripherals (or computers and computers) are well known. It avoids the necessity of designing a large number of special-purpose computer and peripheral specific interfaces. Normally, only one peripheral-independent computer interface need be designed for each computer type and only one computer-independent peripheral interface need be designed for each peripheral type. Furthermore, an intermediate interface becomes almost mandatory when computers and peripherals (or computers) are connected via a communications network. The DIM interface [MORL83] has been especially designed to facilitate computer-peripheral and computer-computer transfers either directly or via MININET.

3.1 INTERFACE REQUIREMENTS

Early in the specification of MININET, it was necessary to adopt a flexible and economical standard interfacing technique that was compatible both with the requirements of its application areas (e.g. laboratory instrumentation, real-time audio processing, process control, etc.) and with the transparency, cost and speed goals of the network (Section 1.2). The requirements of the interface can be summarized as follows:

(1) A single interface must be able to support 16-bit transfers in both directions and, in addition, the transfer of control information in either direction. That is, in terms of conventional computer terminology, a single interface should support "read data", "write data", "read status" and "write command" operations.

(2) Especially in process control and medical applications, it may be necessary to isolate a maverick device which is threatening to disrupt the entire system by some form of unsociable behaviour. This, therefore, precludes the use of a bus system, such as

CAMAC [ESON72] and the IEC-625 bus [IEC 79], because there only has to be a short on a transfer control line, within one device, to halt all operations on the bus, or on a data line to corrupt the data transfers between any devices on the bus. An interface between individual devices allows each device to be isolated. Furthermore, it allows the network to apply flow control to each device individually, thus avoiding the danger of one user flooding the network with its data.

(3) Transfer rates should be controlled by asynchronous handshake signals in the interface to enable data throughput that is as fast as can be comfortably accommodated by both the sender and receiver. This freedom, however, may well be qualified by some relatively long timeout period to detect if the other party is powered down or faulty.

(4) Protection against both internal and external electrical interference should be provided to avoid false transfers, lost transfers or corruption of the information during transfer. Handshake interfaces are notoriously prone to false transfers and to transfer cycles being aborted early, due to impulsive noise appearing on the handshake lines. Thus, special attention should be given to the protection of these lines. Whether or not the data itself should be encoded for error protection across the interface is less certain, as most instrumentation systems and computer input-output systems have no such protection. Computer-computer block transfers would usually include end-to-end error protection of the entire block to cover all parts of the data's journey. Thus, the extra complexity of error encoding in the interface does not seem justifed for most applications. Nevertheless, there may be some very sensitive applications where, at least, a parity check should be made. Consequently, the interface should provide the **option** of including a parity bit with the data.

(5) The maximum transfer rate of the interface should be greater than the maximum user throughput requirements. However, it should not be so fast as to require any exotic logic technology in the interfaces. This requirement implies a maximum throughput in the 10-20Mbps range. This speed is more than sufficient for most instrumentation applications. Of course, in an unsympathetic noise environment, the maximum transfer rate may be reduced in order

to improve the dynamic noise immunity, by allowing longer validation intervals (Section 3.2.1).

(6) Under normal conditions, the interface should be able to operate over distances of, at least, 10m at maximum transfer rates and up to 30m at reduced transfer rates. These distances are as great as, if not greater than, the maximum distances expected within one application area. Wider separations would be serviced by a communications system such as MININET.

(7) The interface should operate in such a way that a transparent communications network can be inserted between two devices, that had been communicating directly, without any change to the interface or higher layer protocols. The most important consequence of this requirement is that the interface cannot support directly elicited responses. That is, there can be no equivalent to the read strobe found in most computer memory and I/O buses because the propagation delay through the network would, most likely, be much longer than the maximum read cycle time the computer could tolerate. The equivalent of directly elicited responses can be achieved by sending a control message requesting the desired data. In order to be compatible with computer buses, which expect to read data and status directly with minimum delay, the computer interface must contain data and status registers which are updated from the peripheral via the interface.

(8) The cost of the interface should be low, commensurate with the relatively inexpensive peripherals that it is interfacing. This constrains many of the above requirements. Without cost constraints, it would be possible to design a very fancy interface with a very fancy price tag which, for the last very good reason alone, would not be used! Consequently, the interface has to do its job as quickly and efficiently as possible while remaining cheap and easy to implement.

3.2 INTERFACE SPECIFICATION

DIM is fundamentally a symmetrical interface with the operation of the two sides of the interface being almost identical. However, there are some differences between the two sides associated with line assignment

and arbitration. For this reason, one side of the interface is termed the ***master*** and the other the ***slave***. Negative logic convention is used for all lines in the interface with the line terminations attempting to pull each line to a high (i.e. false) level. Therefore, the driver's low output state must be active, while the high state may be active or passive. Further information on the electrical and mechanical characteristics of the interface can be found in [MORL83].

3.2.1 The Basic DIM Interface

The basic DIM interface consists of 22 signal lines comprising 16 bidirectional data lines, a data/control shift line, master handshake and slave handshake control lines, and a dominance line.

(1) Data Transfers

Sixteen parallel ***data lines (DDL0–DDL15)***, numbered in a bigendian sense [COHE81] (i.e. bit 0 is the most significant), are used to perform the required 16-bit transfers. Of course, devices do not have to use all of these lines. For example, terminals usually transmit characters along the 8 least significant lines while ADCs, having a resolution less than 16 bits, usually use the most significant lines. The lines are bidirectional to avoid the unnecessary wires, connector pins and electrical buffering components required if two unidirectional sets of data lines are used.

Transfer of data between a computer and a peripheral is usually divided into command or status information on the one hand, and actual end-user data on the other. Different registers are associated with each class of transfer. The ***data/control shift line (DDC)*** qualifies the data lines to distinguish between these classes during a transfer. If this flag is set, the data lines contain normal end-user information (***data class***). If DDC is not set, the data lines contain control information (***control class***). Like the data lines, DDC is bidirectional. This enables a single DIM interface to handle the four basic computer-peripheral operations: read-data, write-data, read-status and write-command.

(2) Transfer Control

Two pairs of transfer control lines are provided in the interface. The master handshake lines, ***master interrupt (DMI)*** and ***master***

acknowledge (DMA), are used to control the transfer of data from the master to the slave. The slave handshake lines, ***slave interrupt (DSI)*** and ***slave acknowledge (DSA)***, are used for transfers from the slave to the master. When a device wishes to transfer data (of either class), it sets its interrupt line true (DMI if it is the master or DSI if it is the slave) to indicate it has data available. After the other device has detected this interrupt and is ready to receive data, it replies by setting its acknowledge line true (DMA if it is the slave or DSA if it is the master). Once the original device has detected the acknowledgement it then, and only then, enables its data and data class transmitters (17 bits in all), thus placing the data to be transferred onto the interface lines. At the same time, it sets its interrupt line false. When the receiver has detected the removal of the interrupt, it first waits for any reflections and crosstalk on the interface cable to die down before quietly accepting the data (usually by loading it into a buffer register). It then sets the acknowledge line false. When the sender detects the removal of the acknowledgement, it disables the data and data class transmitters and the transfer cycle is complete.

If a noise impulse appears on an interrupt line, there is a danger that it could cause a false transfer of nonsensical or duplicate information, or cause the corruption of data passing in the opposite direction. Similarly, a noise impulse on an acknowledge line could result in the loss of a transfer and/or corruption of data passing in the opposite direction. The designers of the British Standard Interface (BS4421) [BSI 69] recognised this danger [DAVI73] and protected their interface by the use of low-pass filters and large hysteresis receivers on the handshake control lines. An alternative method, using ***validation timeouts***, has been adopted for use with DIM. This method is more suitable for implementing in an LSI chip than that using low-pass filters. Logic state changes of a control line are not accepted until it has stayed in its new state for a fixed validation interval. If, at any time during this interval it returns to its original state, then it is ignored. If, subsequently, it goes to its new state again, the validation timing starts again from zero. Any noise impulses of width less than the validation interval are therefore rejected, no matter how close together they occur, and irrespective of their magnitude.

The noise rejection characteristics of the validation timeout approach may be compared with that of a first-order low-pass filter, by

calculating the impulse height required for acceptance as a function of the pulse width. In order to make the comparison equitable, the time constant of the filter, τ_0, is set so that the delay to a normal signal transition is the same as the validation interval, t_0. That is

$$\tau_0 = \frac{t_0}{\log \frac{\Delta Vs}{\Delta Vs - \Delta Vth}} \qquad (3:1)$$

where ΔVs is the normal signal swing and ΔVth is the swing required to reach the receiver threshold level. Using a rectangular pulse of height, Vp, and width, τ, as the model of a noise impulse, the noise immunity of the filter is given by

$$Vp = (1 - e^{-\tau/\tau_0})^{-1} \Delta Vth \qquad (3:2)$$

while that for the timeout circuit is

$$\begin{aligned} Vp &= \infty, \quad \tau < t_0; \\ &= \Delta Vth, \quad \tau \geq t_0. \end{aligned} \qquad (3:3)$$

Perhaps a more realistic noise impulse model is a decaying exponential function of peak height, Vp, and time constant, τ. For this model, the noise immunity of the filter circuit can be found by equating the peak value of the convolution of the noise impulse and the filter impulse response to the threshold voltage. Solving for the noise immunity,

$$Vp = \frac{(\tau_0/\tau - 1)\ \Delta Vth}{(\tau/\tau_0)^{\tau/(\tau_0 - \tau)} - (\tau/\tau_0)^{\tau_0/(\tau_0 - \tau)}}, \quad \tau \neq \tau_0; \qquad (3.4)$$

$$= e\ \Delta Vth, \quad \tau = \tau_0.$$

Similarly, for the timeout circuit the noise immunity to the exponential impulse model is given by

$$Vp = e^{t_0/\tau}\ \Delta Vth \qquad (3:5)$$

Using (3:2) and (3:3), the performance of these circuits for a rectangular noise pulse model is shown in Figure 3.1 where $\Delta Vs = 1.5\ \Delta Vth$. The performance of the low-pass filter circuit is labelled "L.P.F." and that of the validation timeout circuit is labelled "V.T.O.".

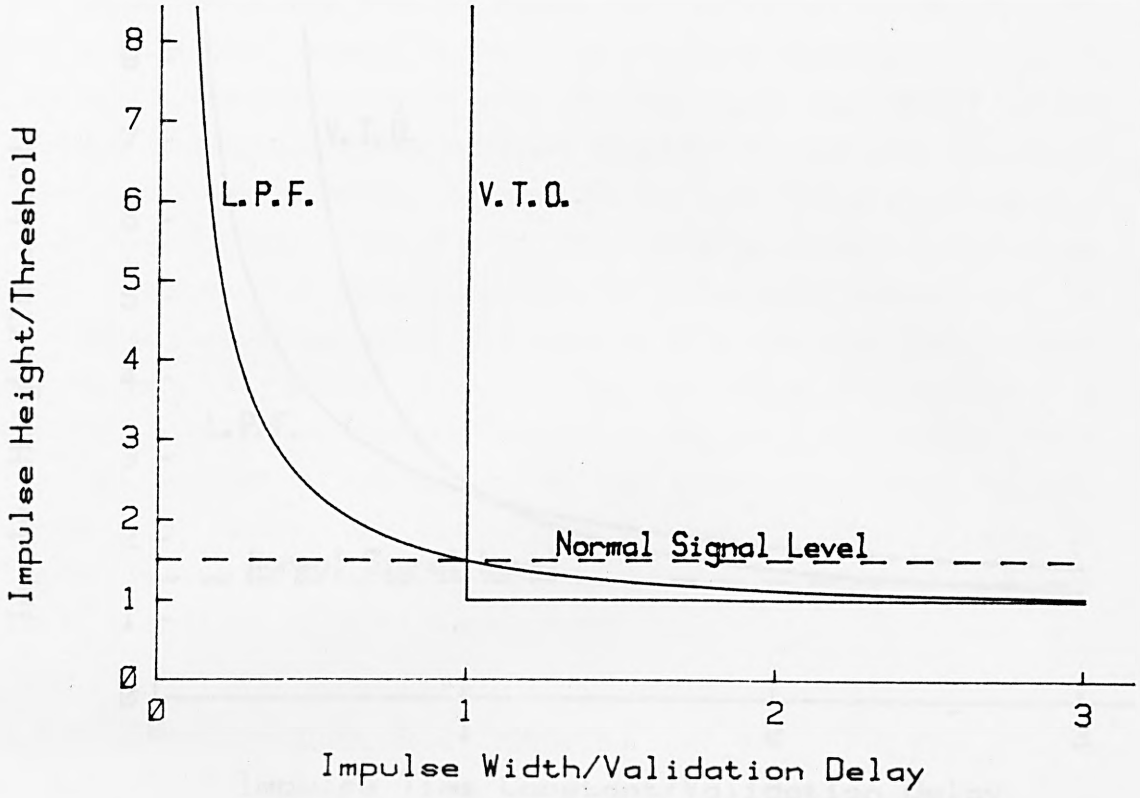


Figure 3.1: Noise Immunity to Rectangular Noise Impulses

Similar plots for the decaying exponential noise pulse model, using (3:4) and (3:5), are shown in Figure 3.2. From both plots, it can be seen that the validation timeout approach has superior immunity against short noise pulses of width less than about t_0, while the filter circuit is slightly better for longer pulses. In addition, a series of closely spaced noise spikes cannot "pump-up" the timeout circuit as they can with the filter circuit, so reducing the latter's immunity.

Leading and trailing edges of the interrupt and acknowledge lines must be validated. Consequently, there are four separate validations in each transfer cycle. In a noisy environment, the validation interval can be increased, provided that it does not slow the maximum transfer rate required by the application. Note also, that the data lines are not sampled until a settling-time interval has elapsed after the last movement of any line on the interface. Thus, the danger of crosstalk or reflections in the interface cable is, for all practical purposes, completely removed. In the present implementation of the interface, the minimum validation interval is approximately 100-200ns, while the minimum settling time, before the data is accepted, is approximately 300ns. This results

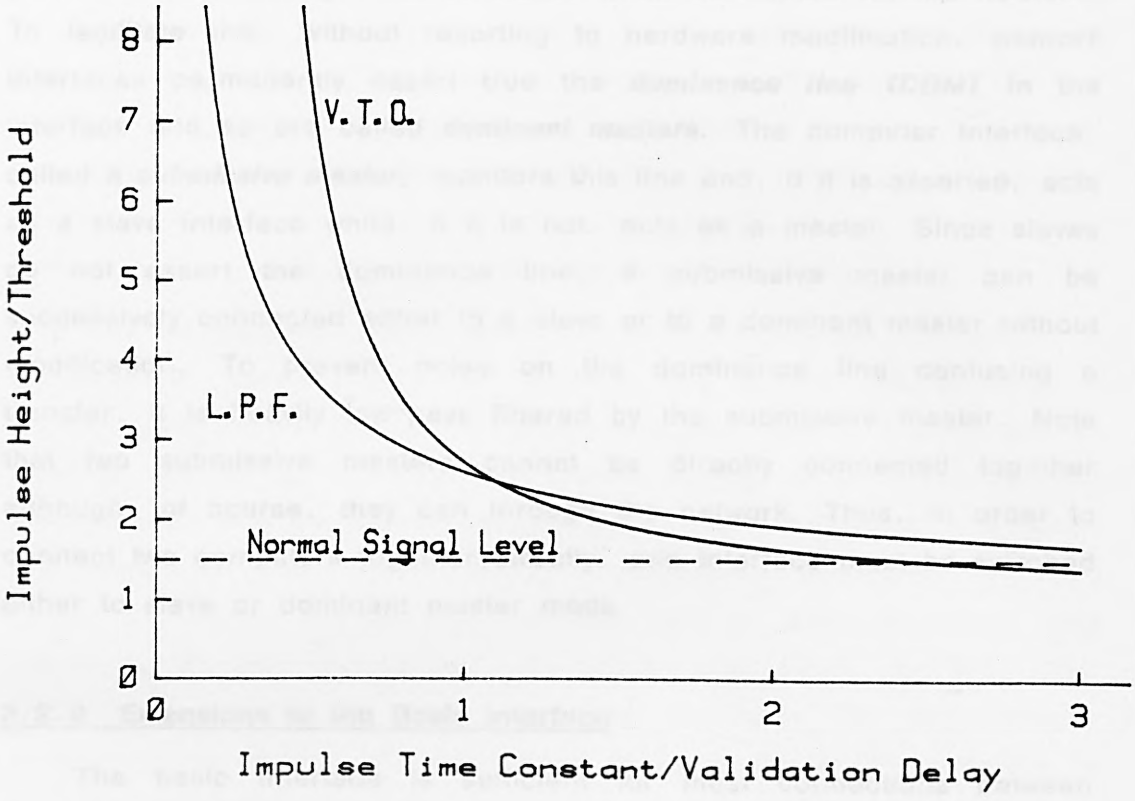


Figure 3.2: Noise Immunity to Exponential Noise Impulses

in the maximum throughput of the interface being approximately one million transfers per second.

(3) Master-Slave Relationship

The master and slave designation of the interfaces principally determines which control lines are used by which party. However, the master interface has one additional function. Since the data lines are bidirectional, it is not possible to perform transfers in both directions at the same time. The master interface arbitrates, on a first-come, first-served basis, which transfer goes first. It performs this by ignoring the master acknowledgement, if it is already asserting the slave acknowledgement, and delaying the slave acknowledgement, if it is already receiving the master acknowledgement. Note that the master designation determines the location of the arbitration logic, rather than any priority of data transfer.

By convention, peripherals use slave interfaces and the network uses master interfaces. It is necessary, therefore, for computer

interfaces to be masters when they are connected directly to a peripheral, while they must be slaves when connected via the network. To facilitate this, without resorting to hardware modification, network interfaces permanently assert true the ***dominance line (DDM)*** in the interface and so are called ***dominant masters.*** The computer interface, called a ***submissive master,*** monitors this line and, if it is asserted, acts as a slave interface while, if it is not, acts as a master. Since slaves do not assert the dominance line, a submissive master can be successively connected either to a slave or to a dominant master without modification. To prevent noise on the dominance line confusing a transfer, it is heavily low-pass filtered by the submissive master. Note that two submissive masters cannot be directly connected together although, of course, they can through the network. Thus, in order to connect two computers together directly, one interface must be switched either to slave or dominant master mode.

3.2.2 Extensions to the Basic Interface

The basic interface is sufficient for most connections between computer and peripherals. However, for some applications, such as certain computer-computer and computer-network connections, there is a need for further qualification of the data transferred. Also, some applications requiring higher data integrity, such as process control, may need some error protection during the data transfer. These requirements are satisfied within DIM by means of address and parity lines respectively.

(1) The DIM Address Lines

Six bidirectional ***address lines (DAL0-DAL5)*** are provided, which are enabled and can transfer information at the same time as the data lines. These lines can be used to provide sub-addressing for links between intelligent devices. For example, two computers can be connected, via a single DIM interface, with the DIM addresses being mapped into the computers' peripheral address spaces to form a number of independent connections between processes in the two machines. Another application arises when a computer is virtually connected to a number of devices via a communications link or a network such a MININET. It is economically undesirable to connect the computer to the

network via separate interfaces for each remotely connected peripheral, as the transfers would first be demultiplexed to each interface and then remultiplexed into the network station. A more attractive solution is to use a single interface between the computer and the network, with the address lines discriminating between the different devices connected. The data would only be demultiplexed when it reached the remote DIM ports connected to the destination devices. Of course, to utilize such an economy, a multiple DIM computer interface is required.

The end-to-end flow control for each multiplexed connection is independent, because each DIM address has its own control class of transfer used by the flow control techniques described in Section 3.3.1. However, as far as the local flow control across the interface between the computer and the network is concerned, the multiplexed connections are treated as one. Consequently, if congestion associated with one Virtual Connection causes the network port to stop accepting data transfers for that connection, it would also block all the other connections multiplexed along that DIM interface. For this reason, multiplexing a number of connections along a single DIM interface is only recommended where the destination devices are associated with the same project or connected to the same destination node.

(2) Parity Control

Two bidirectional parity lines are included in the interface. One, the ***data parity line (DDP)***, provides an odd parity check of the data and the data/control shift lines. The other, the ***address parity line (DAP)***, provides an odd parity check on the address lines. In order to enable interfaces without the parity option to be connected to interfaces that are checking parity, without spurious errors being detected, two additional lines are provided. The ***master parity available (DMP)*** and the ***slave parity available (DSP)*** indicate whether the master or slave interface respectively are generating parity signals. This is a similar facility to that provided in the BS4421 interface [BSI 69]. Of course, a single bit error on this line could result in the detection of a false error or a failure to detect a parity error, because the parity check logic was disabled. This danger is minimized by heavily low-pass filtering the parity available lines. These lines, like the dominance line, are quasi-static and would be expected to change state only when the interface connector is physically moved. Since all the interface signals are asserted-low, which

is the active level, undriven lines are pulled high (i.e. false) by the line termination [MORL83]. Therefore, DMP and DSP are automatically false when connected to a basic master or slave interface which do not use those lines.

3.3 THE COMPUTER-PERIPHERAL CONVENTION (DIM-CPC)

In order to realize fully the advantages of an intermediate interface standard, the design of a computer interface must be independent of the device to which it is connected, whether the device be a terminal, an ADC or another computer. Conversely, the device interface design should be independent of the computer to which it is connected. To this end, only the device interface can contain the device specific circuitry and only the computer interface can contain the circuitry specific to the computer. It is possible to standardize further, as the procedures involved in initializing and maintaining the transfer of information between a device and a computer are remarkably uniform, irrespective both of the device and the computer type. This enables the bulk of the computer interface to be constructed independently, not only of the peripheral, but also of the computer itself.

In order to achieve this independence, it is necessary to specifiy a common initialization, error and flow control protocol and to define the format of the control class transfers between the device and computer interfaces. This is done with the ***DIM computer-peripheral convention (DIM-CPC)***. In addition, the convention allows an intervening communication system such as MININET to report any error condition to the computer interfaces.

This approach facilitates the construction of multiple DIM-computer interfaces like the ***interface processor*** shown in Figure 3.3. Only that part which is concerned with the connection to the computer's peripheral bus need be specific to the particular computer type. This ***computer personality interface*** must conform to the electrical and mechanical standards of the computer bus and may contain logic to map status and command information between the computer's conventional assignments and that of the DIM-CPC.

The central part of the interface contains a data-in, a data-out, a command and a status register for each DIM interface handled. The data-in and data-out registers act as buffers for data class transfers

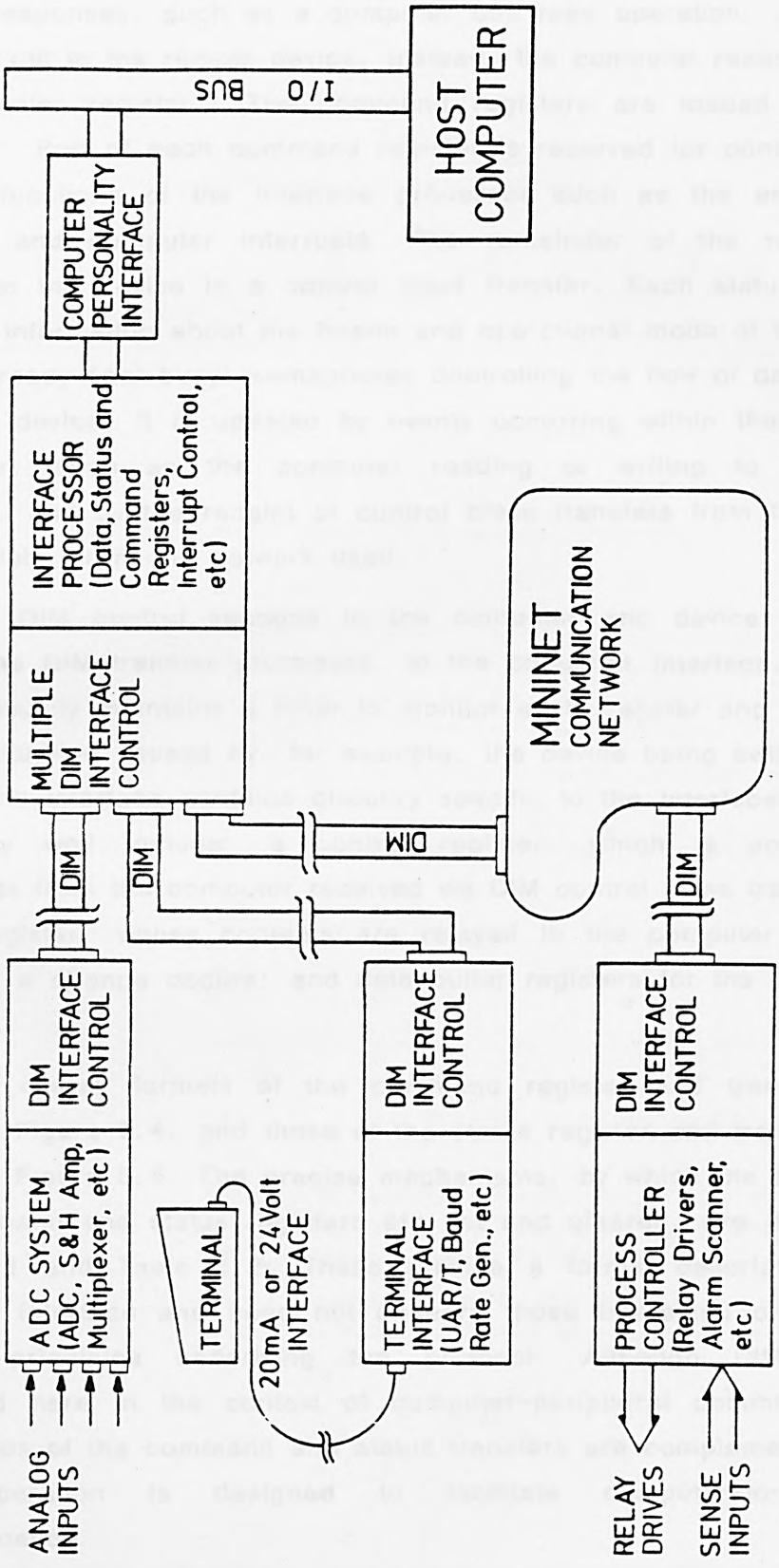


Figure 3.3: Typical Uses of a DIM Interface Processor

between the computer and the DIM interface. (Remember that directly elicited responses, such as a computer bus read operation, cannot be extended out to the remote device. Instead, the computer reads the data in the buffer register.) The command registers are loaded from the computer. Part of each command register is reserved for control of the internal functions of the interface processor such as the enabling of timeouts and computer interrupts. The remainder of the register is relayed to the device in a control class transfer. Each status register contains information about the health and operational mode of the device and the ready (not busy) semaphores controlling the flow of data to and from the device. It is updated by events occurring within the interface processor, such as the computer reading or writing to the data registers, and by the receipt of control class transfers from the device or, possibly, from the network itself.

The DIM control sections in the computer and device interfaces handle the DIM transfer procedure. In the computer interface, the DIM section usually maintains a timer to monitor each transfer and prevent a lockout situation caused by, for example, the device being switched off. The device interface contains circuitry specific to the interfaced device. This may well include: a control register, which is updated by commands from the computer received via DIM control class transfers; a status register, whose contents are relayed to the computer interface whenever a change occurs; and data buffer registers for the data class transfers.

The overall formats of the command register and transfers are shown in Figure 3.4, and those of the status register and transfers are shown in Figure 3.5. The precise mechanisms, by which the bits within the command and status registers are set and cleared, are detailed in Table 3.1 and Table 3.2. These provide a formal description of a DIM-CPC interface and need not concern those interested only in the general principles underlying the protocol. Although DIM-CPC is described here in the context of computer-peripheral communication, the formats of the command and status transfers are complementary and their operation is designed to facilitate computer-to-computer communication.

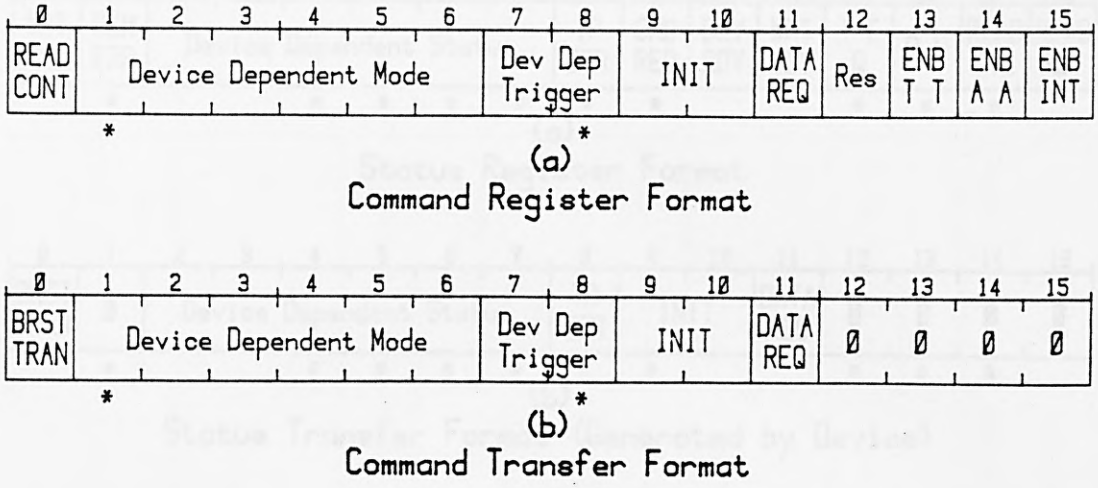


Figure 3.4: DIM-CPC Command Format

BIT	MNEMONIC	DESCRIPTION
0	READ CONT BRST TRAN	Read Continuous (Burst Transfer). Commands the device to output data in burst mode without waiting for a data request acknowledgement from the computer interface.
1-6	Dev. Dep. Mode	Sets mode of operation of the device. Allocation specific to device type.
7-8	Dev. Dep. Trig.	Triggers the device to perform some action which is device-specific.
9-10	INIT	Initialization Commands. See Table 3.3.
11	DATA REQ	Data Request. Requests the device to send one word of data to the computer.
12	Res.	Reserved for future use within the computer interface. Must be zero.
13	ENB T T	Enable Transaction Timer. Allows the transaction "deadman's" timer to run.
14	ENB A A	Enable Auto-acknowledge. Causes the computer interface to send a data request message to the device whenever the data-in register is read by the computer.
15	ENB INT	Enable Interrupts. Causes the computer interface to request a computer interrupt whenever an event requiring computer servicing occurs.

* Use of bits 1 and 8 are restricted to commands to non-intellegent devices only. Otherwise they must be zero.

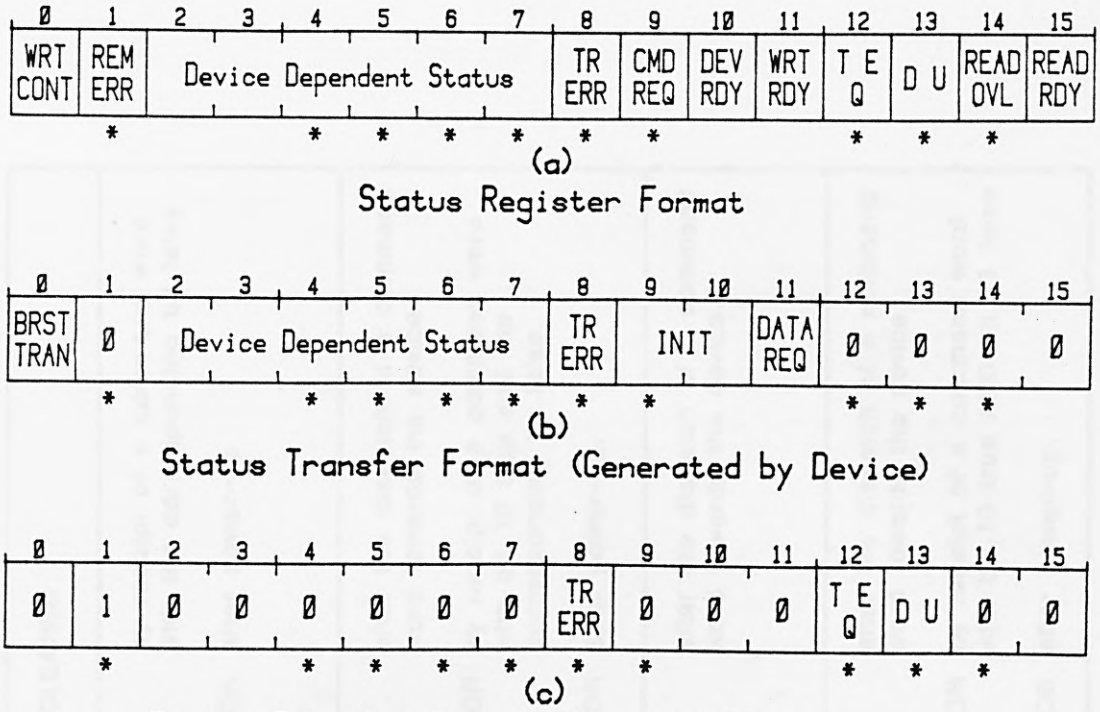


Figure 3.5: DIM-CPC Status Format

BIT	MNEMONIC	DESCRIPTION
0	WRT CONT BRST TRAN	Write Continuous (Burst Transfer). Informs the computer interface that the device can accept data in burst mode.
1	REM ERR	Remote Error. See Table 3.5.
2-7	Dev. Dep.	Allocation specific to device type.
8	TR ERR	Transmission Error. See Table 3.5.
9	CMD REQ	Command Request. Indicates that the device has sent an initialize or block terminate message. See Table 3.4.
10	DEV RDY	Device Ready. Indicates that the device is ready for operation. See Tables 3.3 and 3.4.
9-10	INIT	Initialization Messages. See Table 3.4.
11	WRT RDY	Write Ready. Indicates that the computer can safely write into the data-out register.
11	DATA REQ	Data Request. Requests the computer to send one word of data.
12	T E Q	Transmission Error Qualifier. See Table 3.5.
13	D U	Device Unavailable. See Table 3.5.
14	READ OVL	Read Overlap. Indicates that data has arrived from the device before the previous data had been read by the computer.
15	READ RDY	Read Ready. Indicates that there is new data in the data-in register waiting to be read.

* Exception Condition Bit.

Table 3.1: Command Register Control Functions

BIT	MNEMONIC	SET	CLEARED
0 1-6 12 13 14 15	READ CONT Dev. Mode Res. ENB T T ENB A A ENB INT	by receipt of a command word with the corresponding bit true.	by receipt of a command word with the corresponding bit false OR after power-up
7-8	Dev. Trig.	by receipt of a command word with the corresponding bit true.	after the dispatch of a command word towards the device OR by receipt of a command word with bit 10 true and the corresponding bit false. OR after power-up.
9-10	INIT	by receipt of a command word with the corresponding bit true OR after power-up.	after the dispatch of a command word towards the device.
11	DATA REQ	by receipt of a command word with bit 11 true OR after the data-in register has been read by the computer IF bit 14 of the command register (ENB A A) is set.	after the dispatch of a command word towards the device OR by receipt of a command word with bit 10 true AND bit 11 false OR after power-up.

Table 3.2

Status Register Control Functions

BIT	MNEMONIC	SET	CLEARED after power-up
0 2-3	WRT CONT Dev. Dep.	by receipt of a status word with the corresponding bit true.	OR by receipt of a status word with the corresponding bit false OR by receipt of a command word with bit 9 true AND bit 10 true.
1 4-7 9	REM ERR Dev. Dep. CMD REQ	by receipt of a status word with the corresponding bit true.	OR by receipt of a status word with the corresponding bit false OR by receipt of a command word with bit 10 true.
8	TR ERR	by receipt of a status word with bit 8 true OR by detection of a parity error in a word received from the device or network.	OR by receipt of a status word with bit 8 false OR by receipt of a command word with bit 10 true.
10	DEV RDY	by receipt of a status word with bit 10 true.	OR by receipt of a command word with bit 9 true AND bit 10 true.
11	WRT RDY	IF bit 10 of the status register (DEV RDY) is set, by receipt of a status word with bit 11 true OR receipt of a status word with bit 10 true AND bit 11 true OR, IF bit 0 of the status register (WRT CONT) is set, after dispatch of a data word towards the device.	OR by the computer writing to the data-out register OR by receipt of a command word with bit 9 true OR by receipt of a status word with bit 10 true AND bit 11 false.
12	T E Q	by receipt of a status word with bit 12 true OR by expiration of the transaction timeout interval OR by detection of a parity error in a word received from the device or network.	OR by receipt of a status word with bit 12 false OR by receipt of a command word with bit 10 true.
13	D U	by receipt of a status word with bit 13 true OR by expiration of the DIM timeout interval	OR by receipt of a status word with bit 13 false OR by receipt of a command word with bit 10 true.
14	READ OVL	IF bit 15 of the status register (READ RDY) is already set, by the arrival of a data word from the device.	OR by receipt of a command word with bit 10 true.
15	READ RDY	IF bit 10 of the status register (DEV RDY) is set, by the arrival of a data word from the device.	OR by the computer reading the data-in register OR by receipt of a command word with bit 10 true.

3.3.1 Flow Control

The transfer of information from the computer to the device is controlled by the ***WRT RDY*** and ***WRT CONT*** bits in the status register. After initialization, when it is ready, the device sends a data request message by means of a control class transfer with bit 11 (DATA REQ) true. Bit 0 (BRST TRAN), of the same transfer, determines whether the end-to-end ***handshake transfer mode*** (bit 0 false) or the ***burst transfer mode*** (bit 0 true) is to be used for the write operations. The data request message sets the WRT RDY semaphore in the status register and causes a computer interrupt if enabled. By reading the status register, the computer can sense when WRT RDY becomes true. It can then safely write data into the data-out register. This event immediately clears WRT RDY in the status register. The data is subsequently dispatched, as a data class transfer, through the DIM interface towards the device.

In the end-to-end handshake transfer mode (Figure 3.6a), WRT RDY stays reset until the device signals its readiness to accept another data word by dispatching a data request message to the computer interface which then sets WRT RDY. The process continues with data class transfers from computer to device alternating with control class transfers from device to computer. In the burst transfer mode (Figure 3.6b), WRT RDY is set immediately data is dispatched through the DIM interface and the device sends only the one data request message to start the burst. Thus, data is transferred as fast as the computer can write into the data-out register and the interface can dispatch the data. The device must be capable of accepting data at this rate or a loss of data will occur. The end of the burst is indicated by the computer issuing a block terminate command, which is described in Section 3.3.2.

Note that the computer procedures are identical in both modes of transfer. For this reason, in some implementations, when the computer reads the status register, the ***group error flag (GE)***, which is the inclusive-OR of the exception bits in the status register (bits 4-9 and 12-14), is substituted for WRT CONT in bit 0.

The read process is controlled by ***READ RDY*** in the status register and ***DATA REQ, ENB A A*** and ***READ CONT*** in the command register. If READ CONT (bit 0) is not set by the computer, the data transfer from the device to the computer interface operates in handshake mode. If the

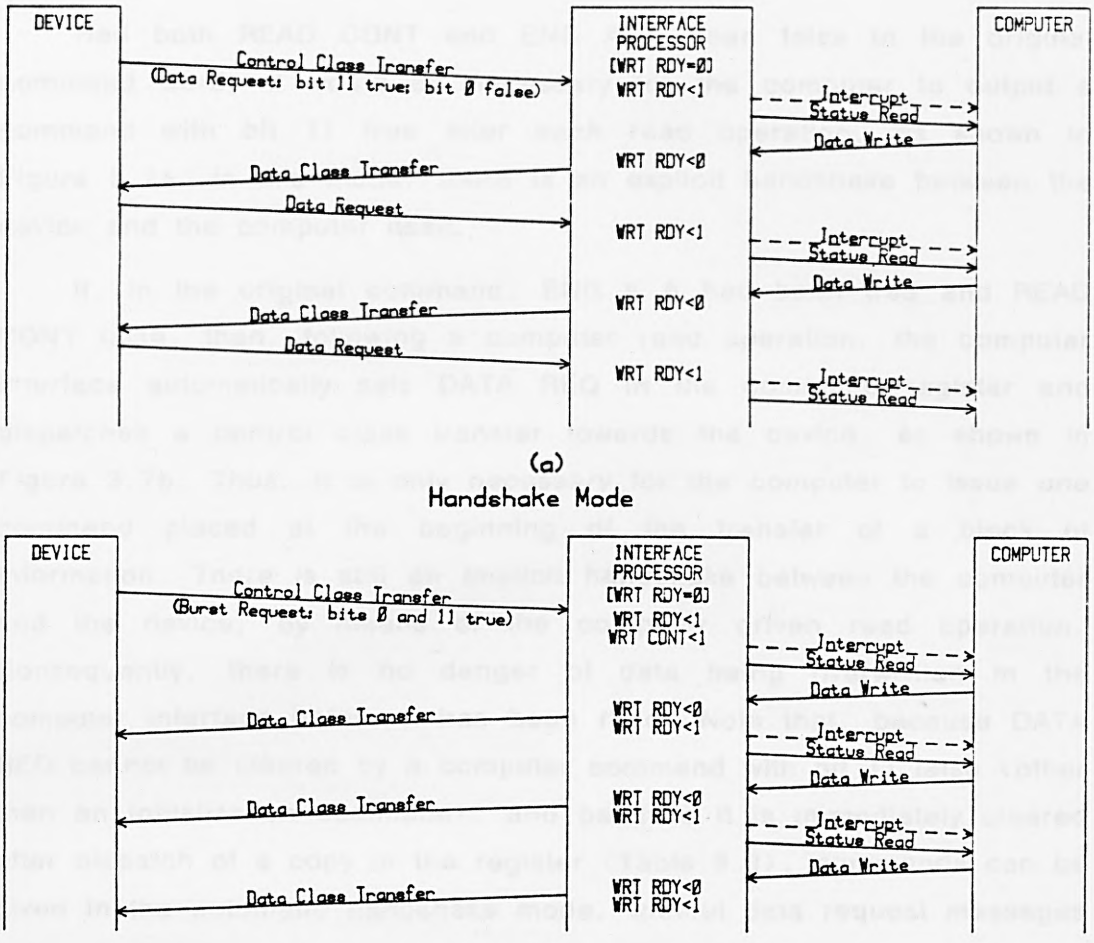


Figure 3.6: Write Procedures

computer sets READ CONT, the data transfer operates in burst mode. ENB A A (bit 14) enables the interface to acknowledge automatically the receipt of data, by sending a data request message **after** the data has been read by the computer.

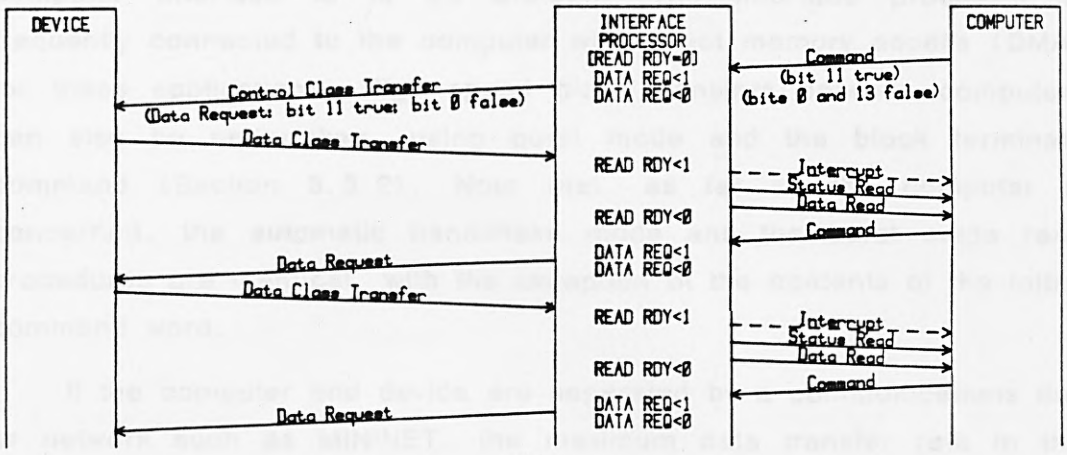
In order to start a read operation, the computer writes a command word with bit 11 (DATA REQ) set. This is dispatched to the device, whereupon the bit is reset in the command register. After receipt of this data request message, the device sends data to the computer. Receipt of the data class transfer by the computer interface causes bit 15 (READ RDY) of the status register to be set and, if enabled, a computer interrupt requested. The computer can detect when the data has arrived by reading the status register and checking bit 15. If it is set, the

computer can then read the data-in register, which causes READ RDY to be reset.

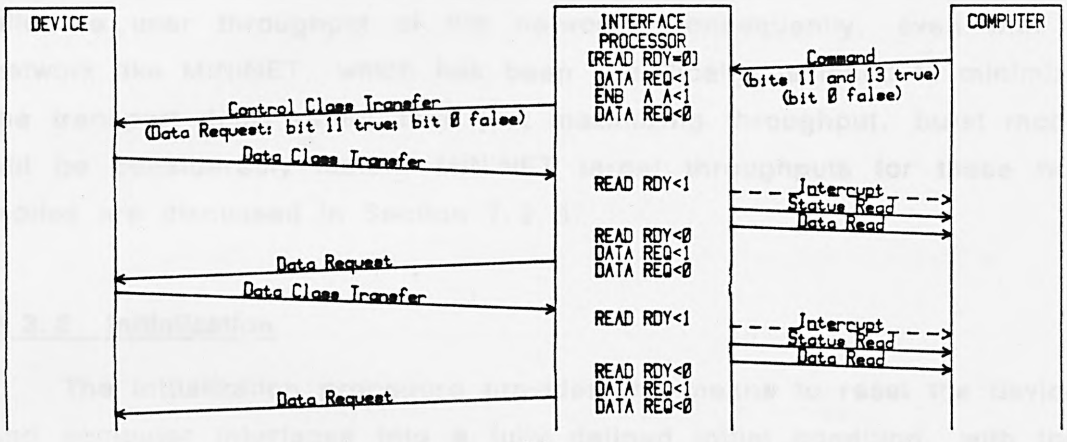
Had both READ CONT and ENB A A been false in the original command word, it would be necessary for the computer to output a command with bit 11 true after each read operation, as shown in Figure 3.7a. In this mode, there is an explicit handshake between the device and the computer itself.

If, in the original command, ENB A A had been true and READ CONT false, then, following a computer read operation, the computer interface automatically sets DATA REQ in the command register and dispatches a control class transfer towards the device, as shown in Figure 3.7b. Thus, it is only necessary for the computer to issue one command placed at the beginning of the transfer of a block of information. There is still an implicit handshake between the computer and the device, by means of the computer driven read operation. Consequently, there is no danger of data being overwritten in the computer interface before it has been read. Note that, because DATA REQ cannot be cleared by a computer command with bit 11 false (other than an initialization command), and because it is immediately cleared after dispatch of a copy of the register (Table 3.1), commands can be given in the automatic handshake mode, without data request messages being lost before they are transmitted, or duplicate requests being generated.

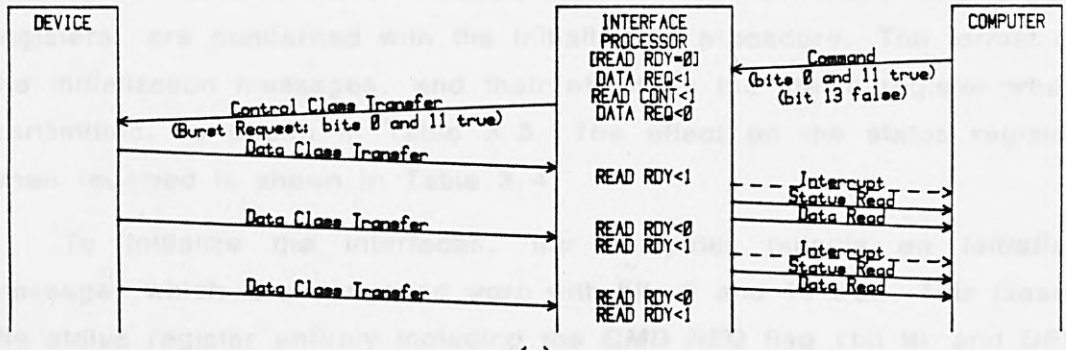
The burst mode of transfer is obtained by setting READ CONT, as well as DATA REQ, in the original command (ENB A A should be reset). The device receives this command, with BRST TRAN true, as a burst request and so sends data continually, without waiting for data request messages, as fast as the data is generated, and the interface or a possibly intervening network can transmit. There is, therefore, a danger that data is not read by the computer before it is overwritten by new data arriving in the data-in register of the computer interface. If this occurs, the error flag ***READ OVL*** (bit 14) of the status register is set to warn the computer. The burst mode of transfer to the computer is commonly used with terminals, which generate data relatively slowly, and, in any case, the character rate from the terminal to the terminal-DIM interface cannot normally be dynamically controlled. Another common application is where a high-speed device, such as a multiplexed ADC, is connected via a network and the trans-network delay



(a)
Non-Automatic Handshake Mode
(READ CONT=Ø and ENB A A=Ø)



(b)
Automatic Handshake Mode
(READ CONT=Ø and ENB A A=1)



(c)
Burst Mode
(READ CONT=1 and ENB A A=Ø)

Figure 3.7: Read Procedures

is too long for the handshake mode. In the latter case, the priority of the computer interface and its software server, as well as the interrupt latency of the computer, can be critical if overlap of data in the computer interface is to be avoided. The interface processor is frequently connected to the computer with direct memory access (DMA) for these applications. High speed block transfers between computers can also be undertaken, using burst mode and the block terminate command (Section 3.3.2). Note that, as far as the computer is concerned, the automatic handshake mode and the burst mode read procedures are identical, with the exception of the contents of the initial command word.

If the computer and device are separated by a communications link or network such as MININET, the maximum data transfer rate in the handshake mode is limited by twice the end-to-end transport delay of the network, whereas, in the burst mode, it is limited by the maximum effective user throughput of the network. Consequently, even with a network like MININET, which has been specifically designed to minimize the transport delay rather than just maximizing throughput, burst mode will be considerably faster. MININET target throughputs for these two modes are discussed in Section 1.2.3.

3.3.2 Initialization

The initialization procedure provides the means to reset the device and computer interfaces into a fully defined initial condition, with the pathway between the two clear of any pre-existing messages. Bits 9 and 10, of the control class transfers and of the command and status registers, are concerned with the initialization procedure. The format of the initialization messages, and their effect on the status register when transmitted, is shown in Table 3.3. The effect on the status register when received is shown in Table 3.4.

To initialize the interfaces, the computer outputs an ***initialize*** message, which is a command word with bits 9 and 10 true. This clears the status register entirely including the ***CMD REQ*** flag (bit 9) and ***DEV RDY*** semaphore (bit 10). If the computer is ready to receive data from the device, bit 11 (DATA REQ) of the command would also be true. If it is not ready, bit 11 of the command word is false, which clears the corresponding bit in the command register (Table 3.1). Thus, any

Table 3.3

Effect of Initialize Commands on Status Register

Message =	Initialize	Initialize Acknowledge	Block Terminate	None
Code (bits 9, 10)	11	01	10	00
DEV RDY (10)	Cleared	Unchanged	Unchanged	Unchanged
WRT RDY (11)	Cleared	Unchanged	Cleared	Unchanged
READ RDY (15)	Cleared	Cleared	Unchanged	Unchanged
Exception Condition bits (1, 4–9, 12–14)	Cleared	Cleared	Unchanged	Unchanged
Non-Exception Condition bits (0, 2, 3)	Cleared	Unchanged	Unchanged	Unchanged

Table 3.4

Effect of Received Initialize Message on Status Register

Message =	Initialize	Initialize Acknowledge	Block Terminate	None
Code (bits 9, 10)	11	01	10	00
CMD REQ (9)	Set	Cleared	Set	Cleared
DEV RDY (10)	Set	Set	Unchanged	Unchanged
WRT RDY (11)	Note 1	Note 1	Note 2	Note 2
READ OVL (14)				
READ RDY (15)	Unchanged	Unchanged	Unchanged	Unchanged
Others (0–8, 12–13)	Note 1	Note 1	Note 1	Note 1

Note 1: Jam loaded from corresponding bit in message.
Note 2: Set if corresponding bit in message is set; otherwise unchanged.

undispatched data request message is overwritten by the initialize command. After the contents of the command register have been dispatched towards the device, bits 9 and 10 are cleared to avoid duplication.

Upon receipt of the initialize message, the device resets itself, clears any error flags (unless the fault condition still exists), and halts any ongoing data acquisition or output. After this process is complete, the device sends a control class message back to the computer. This message contains the new status of the device and has bit 9 false and bit 10 true (***initialize acknowledge***), to acknowledge the initialize command. The arrival of this acknowledgement, at the computer interface, causes the DEV RDY semaphore (bit 10) in the status register to be set, thus informing the computer that the device is ready for operation. Any subsequent status transfers from the device, informing of exception conditions or carrying a data request, must have bits 9 and 10 false. The DEV RDY semaphore remains set, being only cleared by a further initialize command from the computer (Table 2.2).

In the interval between the initialize command and its acknowledgement, while DEV RDY is reset, the WRT RDY and READ RDY semaphores remain firmly reset - despite the arrival of any data or data requests from the device. Thus, any pre-existing data traffic is ignored. However, other status bits, including error flags, can be set and computer interrupts are not suppressed. Consequently, the computer is informed of any error message, which may well originate in an intervening network, concerning the fate of the initialize message.

The device itself can issue an initialize message, which indicates that it is initialized, but it requires a command from the computer as an acknowledgement. (For a non-intelligent device this usually only occurs after power-up.) This message sets bit 9 (CMD REQ) and bit 10 (DEV RDY) in the status register of the computer interface. The WRT RDY semaphore is jam loaded with the value of bit 11 (DATA REQ) of the status message. The computer must respond with bit 10 of its command word true - normally by sending an initialize acknowledge message. When the computer loads the command register with this message, the exception condition bits and READ RDY in the status register are reset. However, DEV RDY, WRT RDY and the remaining non-exception condition bits (0, 2 and 3) remain unchanged (Table 3.2), so that, if a data or burst request was included with the received initialize

message, they are not lost.

It is quite acceptable for the computer to respond with an initialize instead of an initialize acknowledge command. However, this second initialize message would require an acknowledgement from the device before data transfer could begin. Of course, if both sides responded with initialize messages deadlock would result.

The ***block terminate*** command (bit 9 true, bit 10 false) is used, following the transmission, usually in burst mode, of a block of data. It signifies to the device that the block is complete, and that the device should reply with a further (burst) data request message (bits 0 and 11 true), when it is ready for the next block. The block terminate command automatically clears the WRT RDY semaphore in the status register (Table 3.3), which was previously set by the dispatch of the last data word. Thus, the computer will wait, until it receives the data request message from the device, before transmitting the next block of data in burst mode. Reception of a status word containing a block terminate message sets the CMD REQ flag. If data was being received in burst mode, and then transferred to the computer memory via a DMA controller, the block terminate message would cause the controller to terminate the transfer and interrupt the main processor, since CMD REQ is an exception bit. The computer would then be able to process the data or switch DMA buffers, prior to requesting the next block in burst mode, by sending a command with bits 0 and 11 set. Had the device selected the handshake mode of transfer (by keeping bit 0 of its status message false), the WRT RDY semaphore would not be affected by the block terminate command, since it would be reset already. In this case, the terminate command merely delimits the block for the device.

Once set by the computer, bits 9 and 10 of the command register are only reset after dispatch to the device (Table 3.1). Consequently, the initialize commands cannot be inadvertently lost or duplicated by any subsequent commands, provided that, bits 9 and 10 of these commands are reset. An initialize command has priority over an initialize acknowledge command, in the sense that an initialize command will overwrite an, as yet undispatched, acknowledge message in the command register.

Data, or burst, requests and other status or command messages may be included with an initialize or initialize acknowledge message in

either direction. These overwrite any flow control information transferred prior to the initialization of the two interfaces. For example, at the beginning of the transfer of a block of data from a device in automatic handshake mode (Figure 3.7b), the computer can issue a single command with bits 9, 10, 11 and 14 true. Both interfaces are then initialized and the device informed that the computer is ready to receive data. The device then acknowledges the initialize command and sends the first data word. After the computer reads this, the next data request is automatically returned to the device with bits 9 and 10 false. In order to avoid data being transferred before the initialize acknowledgement has been dispatched, it is important that control class transfers have priority over data class transfers, when being dispatched by either interface. Of course, the communication network being used must handle all transfers on a first-come, first-served basis, quite independently of the data class. Indeed, the initialization procedure depends on the network maintaining a strictly sequential flow of information between the two interfaces.

3.3.3 Exception Conditions

The status register contains two types of status bits, in addition to the three semaphores concerned with initialization and flow control (bits 10, 11 and 15). If any of bits 1, 4-9 or 12-14 are set, an ***exception condition*** exists and the group error (GE) flag becomes true. The exception condition may be an error occurring within the device (such as a paper low condition in a printer), or it may be an error arising out of the DIM interface functions themselves (such as a parity error), or it may be an alarm condition such as an bearing overheating in the plant being controlled by the device. It may not be an error condition at all - such as a block terminate message resulting in CMD REQ (bit 9) being set in the status register. The common characteristic of all these conditions is that the flow of data cannot continue without special action being taken. The form of the recovery procedure must, of course, remain device dependent. However, it typically involves the re-initialization of the interfaces, followed by a second attempt at the data transfer, although frequently the condition requires manual intervention to clear the fault.

The GE flag enables the computer or DMA controller to ascertain whether the device is healthy by testing a single bit. Device service

routines normally check that no exception condition bit is set in the status register, prior to testing the READ RDY or WRT RDY semaphores.

The non-exception condition bits (0, 2 and 3) of the status register do not set the GE flag and so can be set without affecting the data flow. Bit 0 (BRST TRAN) is reserved for use by the flow control procedures (Section 3.3.1), while bits 2 and 3 can be used, for example, to indicate the device's mode of operation.

Whenever a status message arrives, the exception and non-exception bits, of the status register, are updated with the contents of the corresponding bit in the message. The exception to this is bit 14 (READ OVL), which is wholly controlled from within the computer interface. It is necessary for the device to send a status message whenever its internal status changes. This status message may be transmitted together with flow control or initialization messages in the same control class transfer.

An exception condition can be detected and reported by the device, by an intervening network such as MININET or by the computer interface itself. The exception conditions, READ OVL (bit 14) and CMD REQ (bit 9), have already been described. Bits 4-7 are used for the exception conditions that arise from within the device. Obviously, their detailed assignment is highly device dependent. Bits 1, 8, 12 and 13 are used to signal exception conditions that are concerned with the general DIM interface functions and are almost completely independent of the device type. The format of this ***interface error*** group of messages is shown in Table 3.5.

Some of the error conditions are detected by the computer interface itself. If its DIM interface times-out when attempting to output to the device, bit 13 (***D U***) of the status register is set to signify a ***local DIM timeout*** error. This timeout could be due to the interface being physically disconnected or the device powered down or, if connected through a network, the local node could be powered down. If, when the **network** attempts to deliver a word to the device, a DIM timeout takes place due to the device not being connected or being powered down, the network sends a ***remote DIM timeout*** message back to the computer interface.

If a parity error is detected in an incoming transfer, bit 8 (***TR ERR***) and bit 12 (***T E Q***) of the status register are set to indicate that a ***received transmission error*** has been detected. If a transmission error,

Table 3.5

Interface Error Codes

Message	Origin	Status Bits			
		1 REM ERR	8 TR ERR	12 T E Q	13 D U
Local DIM Timeout	Computer Interface	0	0	0	1
Transaction Timeout	Computer Interface	0	0	1	0
Received Transmission Error	Computer Interface	0	1	1	0
Transmitted Transmission Error	Device	0	1	0	0
Remote DIM Timeout	Network	1	0	0	1
Outgoing Transmission Error	Network	1	1	0	0
Incoming Transmission Error	Network	1	1	1	0
Link Down	Network	1	0	1	1
No Error		0	0	0	0

in data travelling in the opposite direction, is detected by the device, it can inform the computer by sending a ***transmitted transmission error*** message. If a parity error is detected by the network, or a word is dropped for any other reason, the network can send an ***outgoing transmission error*** or an ***incoming transmission error*** message, depending on the direction of the damaged transfer. It should be noted that MININET does not use this mechanism for error recovery in its channels, as this is handled within the network quite transparently to the user. This type of error message occurs if a parity error is detected, by the network, in data transferred through the DIM port, or (hopefully very rarely) due to corruption of the data within a network node. In all cases

of a parity error being detected, the damaged word is never delivered. If, due to node or channel failures, the network cannot deliver or receive any data from the device, the computer is informed by means of a ***link down*** message.

All the exception conditions reported by the network are characterized by the ***REM ERR*** flag (bit 1). The network must know whether a particular device is capable of accepting these network error messages - i.e. whether the device is intelligent and conforms to DIM-CPC (at least as far as the exception condition handling is concerned). Separate flags in the VCT (Section 2.5.1) indicate whether errors can be reported to the device connected to the local network port or, alternatively, whether they can be sent to the device connected to the remote port at the other end of the connection. In the case of transmission errors where both devices are intelligent, the error message is sent to the device that should have received the damaged word.

In order to avoid a lockout situation, due to a data or data request word being lost between the device and the computer interface, or due to a fault condition within the device, the computer can request its interface to maintain a "dead-man's timer" by outputting a command with bit 13 (***ENB T T***) true. While this bit is set in the command register, a timer is reset and started whenever a word of either class is dispatched towards the device. The timer is halted whenever: a data or control class message is received from the device or network; data is dispatched in burst mode; or a DIM timeout or parity error is detected by the interface itself. If none of these events occur, a timeout will eventually take place and bit 12 is set in the status register to indicate that a ***transaction timeout*** has occurred.

It can be seen, from Table 3.5, that bit 13 (D U) of the status register is set if there is a local or remote DIM timeout, or the device is unreachable through the network. This flag, therefore, serves to inform the computer that the device is unavailable. Also, bit 8 (TR ERR) indicates that, somewhere, a transmission error has occurred.

3.3.4 Command Structure

The control of the internal operation of the non-intelligent devices is effected by means of the device-dependent portion (bits 1-8) of the

command word. There are two types of command bits. Most affect the mode of operation of the device and they remain in force until the computer explicitly changes the mode. These ***mode commands*** are duplicated every time a command word is sent to the device from the computer interface. These duplications do not have any deleterious effect, as they merely update the control register within the device with the same information that it already contains. Typical of this type of command function are echoplex control of a terminal interface and selection of an external or internal clock in a DAC. Device dependent bits 1-6 are used for mode commands and bit 0 (READ CONT) is also of this type.

The other type of command triggers a single event or sequence of events in the device. For example, a single command may trigger an analog data acquisition system to perform a scan of its input channels. Device-dependent bits 7 and 8 are used for these ***trigger commands.*** The data request and initialization commands are also of this type. Clearly, duplication of a trigger command must be avoided. For this reason, the trigger bits (7-11) in the command register are cleared after dispatch to the device.

Bits 12-15 of the command register are reserved for internal use in the computer interface and are always transmitted as zero towards the device (Figure 3.3). Bit 15 (***ENB INT***) is used to enable computer interrupts. If it is set, a computer interrupt request is generated whenever: data or status information is received by the interface; or when a DIM timeout, parity error or transaction timeout is detected by the interface; or when data is dispatched towards the device if WRT CONT is set in the status register. The use of bit 13 (ENB T T) and bit 14 (ENB A A) has already been described. Bit 12 is reserved for future use within the computer interface and should remain zero.

3.4 OPERATIONAL EXPERIENCE

The DIM interface has been in use at the Polytechnic of Central London and the University of Bologna since the mid-seventies, with interfaces to DEC PDP-11, Perkin-Elmer and Apollo computers. In that time, a large amount of equipment has been constructed using the DIM interface, including high-speed converters for digital audio processing and recording, computer-controlled adaptive filters and an arbitrary

waveform generator, as well as the more usual computer peripherals, such as terminal interfaces, paper tape readers, etc. The interface has also been used for resource sharing between microcomputer development systems and a minicomputer, allowing the microcomputer access to the hard discs and fast printers of the minicomputer system. Construction of most DIM devices has been greatly eased by the use of a standard DIM interface circuit board, which handles the transfer cycle and flow and initialization protocols. It is only necessary, therefore, to design the device-specific circuitry for each device type. As an example, Figure 3.8 shows an exploded view of a 16-bit stereo DAC

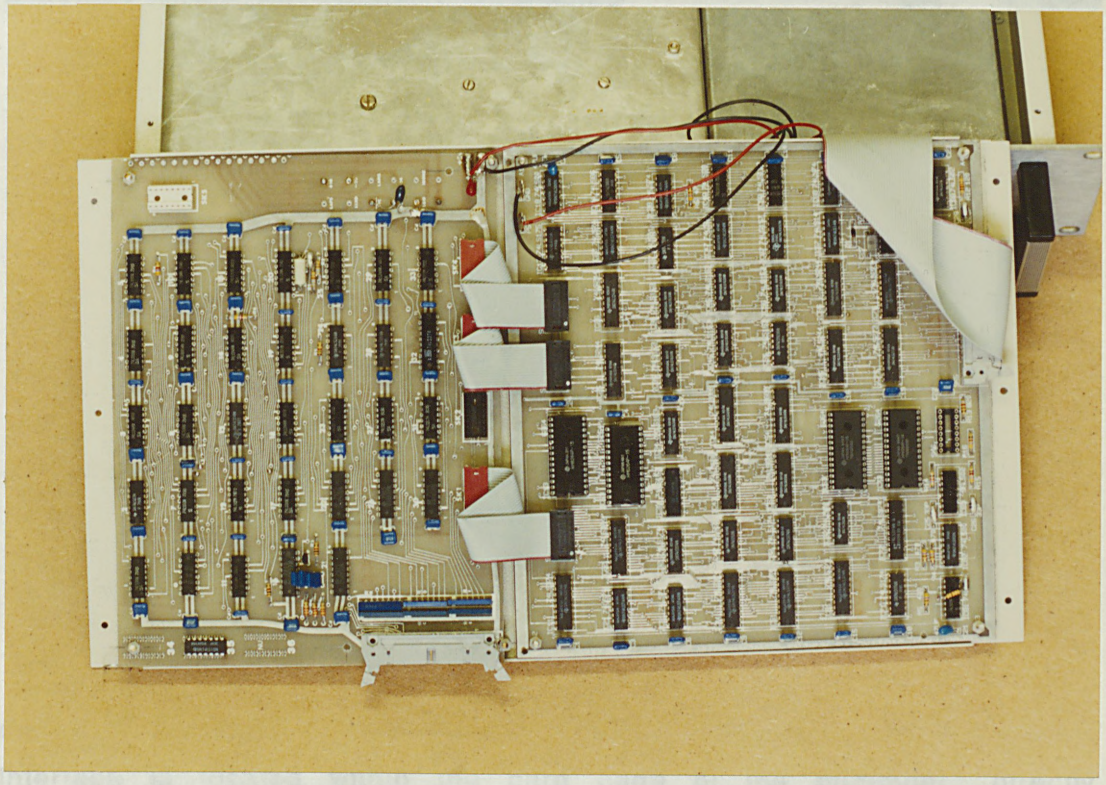


Figure 3.8: Construction of a HI-FI DAC with a DIM Interface

capable of working up to a 50kHz sample rate. The DIM interface circuit board can be seen in the left of the picture. Work is currently underway to replace this board with a custom chip. To the right of the DIM interface card is the device specific board. This contains two 8K x 16-bit buffer memories, which are used to buffer blocks of audio data transmitted through the DIM interface in burst mode (Section 3.3.1), together with control and timing logic, address counters and bus switches. The DAC itself, together with its deglitching amplifiers, is contained in the diecast box located in the upper right of the picture. The DAC is optically isolated from the rest of the device circuitry in

order to eliminate interference from the digital circuitry.

Figure 3.9 shows the implementation of an interface processor

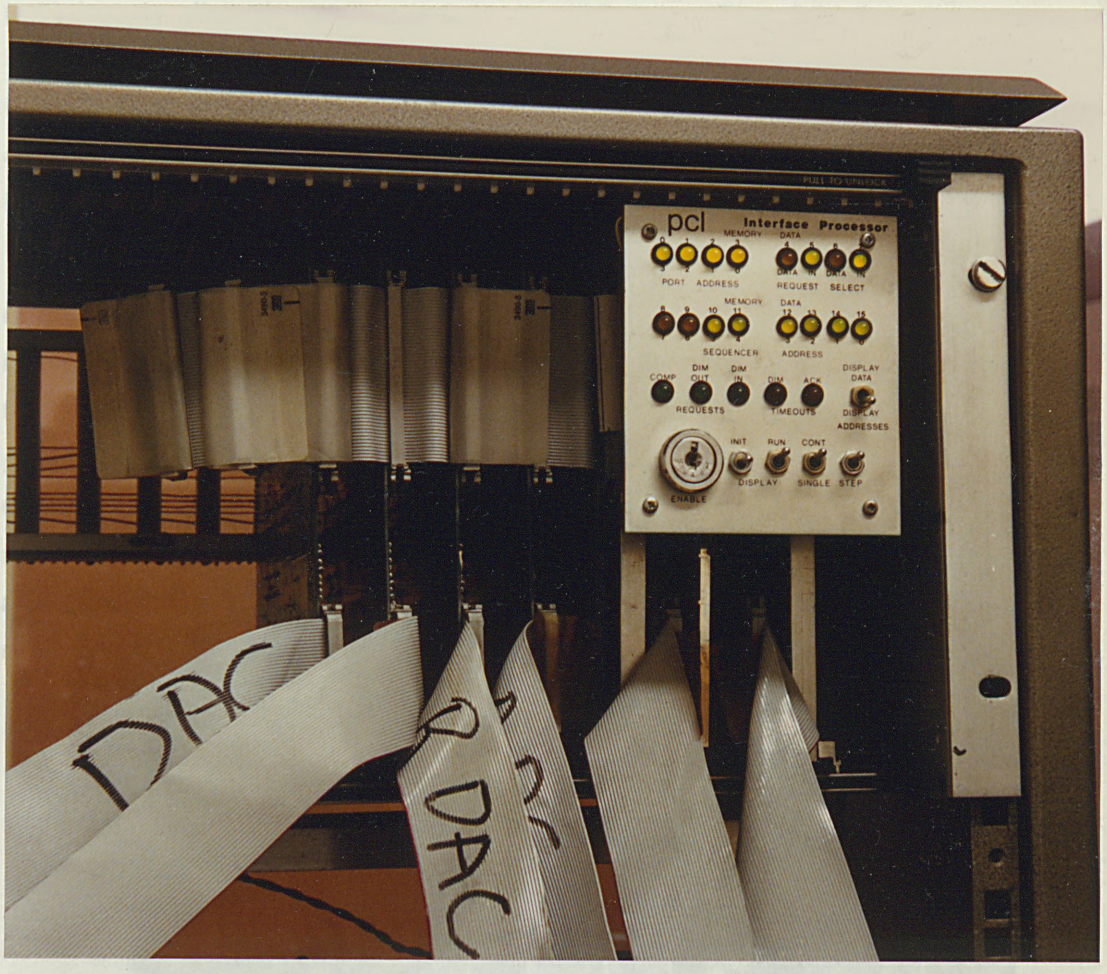


Figure 3.9: Implementation of an Interface Processor

connected to a Perkin-Elmer 3210 computer. The computer personality interface is located within the card cage of the computer, directly connected to its I/O backplane bus. This is connected to the interface processor by means of two 34-way flat cables which can be seen in the lower right of the picture. The interface processor can handle up to 16 DIM interfaces by means of the DIM ports connected immediately to the left of the interface processor. (In the picture, 4 ports are shown connected to their devices via 34-way flat cables.) The main part of the interface operates as a high-speed FSM, which executes a routine whenever a computer read or write, DIM transfer or a timeout takes place. The main controller is implemented using a microprogrammed, purpose-built microsequencer. The data, status and command registers for each DIM interface are located within a 64 x 16-bit RAM. A special processor, implemented using field programmable logic arrays (FPLAs),

is used to perform the various operations on the status and command registers. The speed of the processor is such that the computer treats the, sometimes quite remote, devices as if they were separate peripherals directly connected to its I/O bus. The DIM interface control is handled by a separate processor. A bank of addressable semaphores is used to indicate when the contents of a data-out or command register are scheduled for DIM transmission. The interrupt requests are handled in a similar manner by a different bank of interlock semaphores.

With the advent of affordable custom integrated circuits, the economic justification for the **multiple**-port interface processor is much less. Figure 3.10 shows a Multibus (IEEE-796 [IEEE83]) card providing two DIM interfaces. Each interface uses a single-port interface processor implemented on a CMOS gate array. (Only one of these 64-pin DIL packages is shown inserted in the picture.) The bulk of the rest of the circuit board is taken up with Multibus address comparators and transceiver chips to provide the relatively high drive currents required by DIM and Multibus.

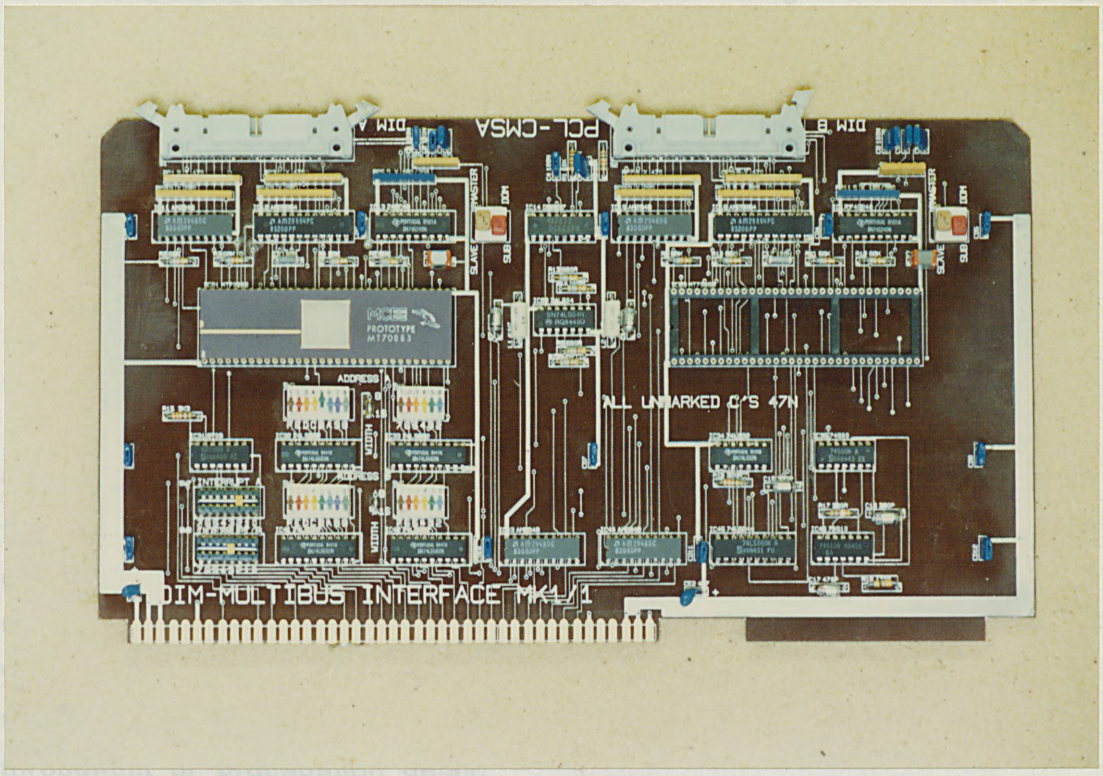


Figure 3.10: A Dual DIM-Multibus Interface

Chapter 4

THE ROUTING ALGORITHM

4.1 REQUIREMENTS

The objective of the routing management algorithm is to find the "best" routes from each node to every other reachable node in the network. The best route or ***path*** between two nodes is defined as the route which has the minimum "cost" compared with all other possible routes. This cost is computed as the sum of the costs of each individual hop along the route and will be referred to as the ***distance*** along that path between the two nodes. Each route should be optimized individually. Note, that this does not necessarily produce a set of routes which is **globally** optimum in the sense of minimizing the total cost of all the routes in the network. In fact, optimizing globally is not only difficult to implement, but also can lead to certain path costs being made unnecessarily high in the interests of the average common good [MCQU77]. Such a proposal would be very unfair.

Each hop is identified as a link defined, in Section 2.3.3, as a pathway from one node to an adjacent node via a channel. Remember that, for multi-node channels, several links are logically multiplexed through the same channel. There are a number of possible methods of defining the cost of each hop (the ***link weight***). It could be the same value (e.g. unity) for all links regardless of the speed or length of the link, or the traffic levels in each link. Such a metric has the advantage of simplicity and results in a algorithm which attempts to minimize the number of hops along each path. However, if there are significant variations in the performance of different links, it is advantageous to weight each link with a hop cost which could be a measure of the link throughput or propagation delay.

Since the network is required to minimize delay rather than maximize the throughput (Section 1.2), link weights should be proportional to an estimate of the ***hop delay***. This is defined as the period between a packet arriving at the Channel Service boundary in the node and it arriving at the same point in the next node. It consists of

two components: one being the time spent queuing for the channel and the other being the channel transit time including any buffering within the channel controllers. Provided that the output switch processor (Figure 2.18) is not limiting throughput, the latter component of the hop delay is inversely proportional to the channel speed for a given channel protocol. Note, that a half-duplex protocol generally provides less delay than a full-duplex channel with the same throughput. If the routing algorithm is ***connectivity-driven***, the link weights do not change dynamically. Instead, an a priori estimate of the queuing delay is used. Hence, routing changes only occur if the connectivity of the network changes (i.e. node or link outages or restoration). It would be quite possible to maintain different hop costs for different destinations. With a ***traffic-driven*** routing algorithm, the hop cost adapts dynamically to changes in traffic conditions as well as the link availability. This requires some sort of ongoing measurement of the delays being experienced by the traffic through each link.

At first sight, some form of traffic-driven metric appears to be desirable for use in MININET. However, there are a number of reasons why the algorithm should only be connectivity-driven.

(1) When all traffic to a particular destination is switched onto a new channel by the routing algorithm, the length of its queues and hence its hop delay will increase. This could well make the previously used channel appear, once again, more attractive causing the routing algorithm to switch back the traffic to the original channel. The consequent increase in traffic on the original channel would trigger a repeat of the first routing change. Thus, instability could well result. Traffic bifurcation cannot be used because of the danger of sequence errors violating the requirements of the Packet Delivery Service (Section 2.4). Incremental changes in route cannot be made when Virtual Connections are established and removed. This is because packets in intermediate store-and-forward nodes are classified purely on the basis of their destination node address, regardless of their Virtual Connection or even their source node. In any case, routing changes would be implemented relatively slowly at a rate dependent on the user opening and closing connections.

(2) In a large network, with many independent users, the aggregate traffic **may** approximate to a quasi-stationary stochastic process,

where statistics gathered from the (recent) past provide a reasonable estimate of future requirements. However, in a small network serving the type of environment for which MININET has been designed, the assumption of traffic ergodicity is highly suspect. There are fewer users, who, by their nature, tend to be intermittent and bursty in their use of the network. Furthermore, since the users are frequently co-operating in the control of linked industrial processes or scientific experiments, they cannot be treated as statistically independent sources. In fact, even with wide area networks, there are real problems in obtaining a meaningful traffic-driven metric. The simplest method is to take a instantaneous sample of the total length of the output queues for each channel and use this, perhaps with the addition of a fixed bias term, as the link weight. This was the method used in the old ARPANET routing protocol [MCQU78]. However, the random fluctuations of the queue lengths cause excessive variations in the link weights. In order to reduce the variance of the delay estimates, the queue lengths could be averaged in some way. While heavy averaging produces a more reliable estimate in a steady state situation, it will respond very sluggishly to any change in the traffic pattern.

(3) It will be shown, in the development of the routing protocol, that the maintenance of intrinsic sequentiality, when routing changes are made, results in unavoidable delay to packets while the old path is flushed. Thus, the effort of attempting to follow an ever changing optimum path may well be more costly than staying with one which is slightly suboptimal.

The link weights would be set initially from a knowledge of the type and speed of the channel through which it passes. This latter information is obtained from the channel controller. Note that, notwithstanding the arguments outined above for fixed link weights, it should be possible for the weights to be adjusted by management entities external to the routing algorithm such as the operator. This manual intervention may well be based on knowledge, external to the network, of future traffic patterns.

The routing algorithm must automatically, and as rapidly as possible, adapt to topological changes in the network, including the

failure and recovery of nodes and channels. Note, that failure and recovery of nodes can be treated as the failure and recovery of all the links to that node, as far as the routing algorithm is concerned.

The algorithm must operate with no a priori knowledge about the topology of the network. The only node-specific information available to the routing manager is the node's own address and the link weights. This implies not only that a newly powered-up node must dynamically find routes to all other nodes in the network, but also that all the other nodes must learn about the new node and find routes to it. Furthermore, the algorithm must be non-centralized in that it should not be dependent for its operation on any central network control centre.

In most routing algorithms, the sequentiality of the packets is not guaranteed following a route change. A sequence error is most likely to occur following the recovery of a node or channel when the new route is considerably shorter than that previously used. The algorithm for use in MININET must guarantee that a route is changed without any sequence errors occurring. Note that this implies, but goes much further than, the requirement that it never routes packets in a loop (loop freedom).

4.2 EXISTING ROUTING ALGORITHMS

4.2.1 Taxonomy

Over the years there have been many attempts to produce a taxonomy of routing techniques [BOEH69], [RUDI76], [MCQU77A], [DAVI79] and [SCHW80]. Figure 4.1 shows a general classification scheme largely based on [DAVI79]. Most routing methods maintain a ***routing directory*** or table in each node. This table is used on a packet-by-packet basis to determine the next link that the packet is to travel.

There are some routing methods, however, that do not require a directory. With *flood routing* [BOEH69], an arriving packet is re-transmitted on all the other links with the exception of the incoming link. Obviously, the network is soon flooded with copies of the packet. In order to stop the process continuing ad infinitum, two methods may be used. In one method, each packet contains a hop counter, which is initially set to a number greater than the number of hops between the most distant node pair in the network. At each hop, the counter within

Figure 4.1: Classification of Routing Techniques.

each packet is decremented. When it reaches zero the packet is destroyed. The other method uses some form of unique identification (e.g. a serial number) to enable each node to remember when it has retransmitted a particular packet. Subsequent reception of the same packet will not cause a duplicate retransmission of that packet. While very wasteful of network resources and increasing the danger of congestion, this method is, at once, simple, robust and fast. For these reasons, it is often used for the dissemination of control information in the more sophisticated routing algorithms such as the new ARPANET routing protocol [MCQU80].

With *random routing* [PROS62], incoming packets are randomly assigned to an output link regardless of whether the link leads towards or away from the packet's destination. Like flooding, this algorithm is simple and requires no knowledge of the network topology. However, it is also very inefficient and indeterminately slow. Again, a hop counter can be used to kill packets that have been in the network too long. This may be because the destination is unreachable, or merely because the random path of the packet happened not to touch the destination node. *Hot-potato routing* [BOEH64] is similar to random routing except that an incoming packet is assigned to the shortest output queue instead of a queue selected at random.

If a routing directory is used, its contents may remain fixed after being initially loaded when the network was configured or it may be updated automatically so that it adapts to changes in the network characteristics. In the former case, the *fixed directory* may contain only a single route to each destination or it may, in addition, contain one or more alternative routes. The international airline reservation network, SITA [BRAN72] uses fixed routes with alternatives stored in each node for use in the event of a node or line failure. In IBM's Systems Network Architecture (SNA), a total of up to 8 alternative routes can be stored [AHUJ79]. With an *adaptive routing* approach, the routing directory is updated automatically. It can be further sub-divided into isolated, distributed, centralized and hybrid methods depending on the scope of the routing information and the location of the entities responsible for the updating of the routing tables. Note, that the distinction between connectivity-driven and traffic-driven metrics is often confused with the distinction between fixed and adaptive routing. A routing algorithm which responds only to changes in the network topology is still adaptive in this

classification scheme.

With *isolated adaptive routing* the routing decisions are taken on the basis of a pre-loaded routing table, the operability of the output links and the length of the output queues associated with each link. One example of this method is the *shortest queue plus bias* algorithm [FULT72] (also discussed in [DAVI79]), which calculates a figure of merit, for each output link, from the weighted sum of the number of spare slots in its output queue and a bias figure obtained from the routing table. Note that, if the weight of the bias term is set to zero, it becomes a hot potato algorithm, while if the weight of the queue length term is set to zero, the routing becomes fixed. Thus, this method may be thought of as a combination of those two techniques already discussed.

In order to increase the efficiency of the routing algorithm, a more global view of the network status must be taken. One approach is to use *centralized adaptive routing* where status reports containing queue lengths, link availability, etc., are sent from all nodes to a *network management centre (NMC)*. On the basis of this information, the NMC calculates the optimum routing table for each node and then distributes them to the nodes. Note that, by the time status reports have reached the NMC, the routing calculations performed and the routing table updates distributed, the information upon which the tables were derived may well be out of date and the chosen routes no longer the best. TYMNET [RAJA78] is the best known example of the use of this technique.

Instead of the routing update calculation being performed at a centralized site, the computation could be spread over the network, with each node calculating its own routing table on the basis of status information received from other nodes in the network as well as local information. This is *distributed adaptive routing*. It has all the reliability and flexibility of distributed techniques in general. However, the disadvantages of the original ARPANET routing protocol [MCQU78], which is probably the best known example of a distributed adaptive routing algorithm, are often erroneously ascribed to distributed routing algorithms in general (e.g. [DAVI79]). In addition, a distributed algorithm shares the problem of centralized techniques in that the information upon which routing decisions are based is always out of date and consequently potentially erroneous. In fact, even if the entire

network status was available to all nodes, the so-called "magic eye" or "ideal observer" technique, the routing decisions would still not necessarily be optimal because the network conditions could change by the time packets have reached other parts of the network [DAVI79].

Of course, mixtures of these techniques can be proposed. One *hybrid adaptive routing algorithm*, proposed by Rudin [RUDI76], is *delta routing* which is a combination of centralized and isolated adaptive routing. The centralized portion gives the long-term, long-distance, strategic view, while the use of local information (i.e. queue lengths and link availability) enables the algorithm to make fast tactical routing changes in response to changes in the local environment.

Given the requirements on the MININET routing protocol of optimality, adaption to changes in topology and avoidance of centralized control, it is obvious that some form of directory-based, adaptive, distributed protocol is required. Following McQuillan [MCQU80], the operation of a distributed adaptive algorithm may be divided into four functions.

1. A *measurement* process to establish the current cost of each hop.
2. A *protocol* for disseminating this information to the rest of the network.
3. A *calculation* to determine the contents of each routing table.
4. An *implementation* of the packet-by-packet routing using the information stored in the routing table.

The first and last function are, more or less, independent of the others. However, the second and third are interrelated depending on how the algorithm is distributed.

The measurement process has already been discussed in Section 4.1 and a connectivity-driven algorithm chosen with the fixed link weights only being modified by manual intervention. The protocol and calculation process are discussed in the next section.

4.2.2 Routing Protocols

Probably the most well known routing protocol is the original ARPANET protocol [MCQU78] which was designed in 1969. It has been adopted, with some modifications, by a number of other networks

including CIGALE, the communications network for CYCLADES [POUZ74] and the European Informatics Network (EIN) [PONC75]. The algorithm, which is a distributed form of the *shortest chain algorithm* [FORD62], seeks to construct, for each destination, a multi-branched tree rooted at the destination. Each node stores, for each of its links, an estimate of the distance to each destination via that link. From this *network delay table* it selects, for each destination node, the link with the minimum distance and enters this into its routing directory. It also places the corresponding minimum distance into its *minimum delay table*. The size of the network delay table is proportional to N.L where N is the number of nodes in the network and L the average number of links attached to the node. The size of the minimum delay table is proportional to N only. Periodically, (every 2/3s in the case of ARPANET) the node transmits its minimum delay table to its adjacent neighbours and receives their minimum delay tables. These tables update the network delay table after the addition of the appropriate link weight (which is traffic-driven in ARPANET). In CIGALE, the update period is normally very long (20s) but more rapid updates occur when network changes occur [GRAN78].

Although the algorithm will eventually converge to the optimal tree under steady state conditions, transient loops are frequently formed while the algorithm adapts to changes in network topology. A most unfortunate characterisic of the algorithm is that, while "good news" (i.e. link recovery) propagates quickly through the network, "bad news" (i.e. link failure) travels relatively slowly [MCQU78]. This is because, following the failure of its down-tree link, a node will believe that it can reach the destination through its up-tree links with little increase in delay. However, this belief is based on tables that were produced before the failure of the link and the up-tree nodes still believe the optimum route is towards the failed link. Depending on how many hops it takes to reach a part of the original tree unaffected by the link failure, it can take several update cycles for the algorithm to settle. During that time, the resulting loops make any attempt at maintaining packet sequency impossible, as well as worsening the congestion following the link failure. This problem was patched-up, to some extent, by means of a hold-down technique, which introduced inertia into the changes to the minimum delay table [MCQU78]. All in all, however, this protocol does not appear to be a very promising basis for the MININET routing algorithm.

The new ARPANET routing algorithm [MCQU80] largely overcomes these problems by the use of a completely different algorithm. Instead of a distributed calculation attempting to construct separate trees rooted at every destination, each node attempts to construct a tree rooted at itself. It is, therefore, a source based algorithm with, in one sense, a centralized calculation performed in the root node. Every node asynchronously broadcasts its set of link weights to all the other nodes in the network using flood routing. This raw information forms the database within each node, that allows it to calculate the tree with itself at the root.

The algorithm, actually used to construct this tree, is an extension of Dijkstra's *shortest path first* algorithm [DIJK59]. Starting with the root, the tree is constructed by adding the node closest to the root and continues by adding the next nearest node and so on, until the furthest node has been included. The next nearest node is selected, at each incremental stage, by maintaining a list of all nodes adjacent to, but not yet part of, the partially constructed tree. Since the minimum distance to the root, from every node within the tree, is known, the current shortest distance for the nodes on the list can be found by adding the appropriate link weight. For the case of a node having more than one neighbour in the tree, the minimum distance is taken. After selecting the nearest node in the list and adding it to the tree, the list is updated by adding any new adjacent nodes and updating their minimum distance. This process is repeated until all nodes have been included in the tree. The routing directory is formed by noting the first branch (i.e. the root's output link) leading to each destination node in the tree. In the ARPANET implementation, the algorithm is extended to allow incremental modification upon the reception of a new set of link weights. The storage requirement of this algorithm is proportional to N.L. The routing message size is proportional to L but, unlike the old algorithm where only one message per update cycle is transmitted along each link, N messages are required to broadcast information about all nodes and their links. While less prone to looping than the original ARPANET algorithm, this protocol does not guarantee freedom from loops. This is because, in a dynamic situation, there is no guarantee that the trees calculated by different nodes are consistent. In addition, a special flood routing protocol is required to broadcast the link weights to all nodes.

As far as the requirements for MININET are concerned, a much

more applicable routing algorithm has been developed by Merlin and Segall [MERL79]. Their algorithm converges to the optimum routes very rapidly, guarantees loop freedom at all times and handles, en passant, the network initialization problem. It is distributed and requires information to be exchanged only between adjacent nodes so that NTAN messages can be used. Unfortunately, it does not maintain intrinsic sequentiality. However, it provides a basis upon which a sequential algorithm can be designed. Like the original ARPANET algorithm, it constructs a tree routed at the destination or ***sink***. Successive update cycles of the protocol minimize the distance from each node to the sink.

Merlin and Segall present two versions of their algorithm. The basic protocol cannot handle certain changes of topology such as node or channel failures. The extended protocol can handle single or even multiple failures. The algorithm operates independently for each sink. Each node has a ***routing link*** (connected to the *preferred neighbour* in the terminology of [MERL79]). These routing links can be thought of as directed arcs which, with the nodes, form a tree directed towards the sink as shown in Figure 4.2. This tree spans the network. The nodes at the tips of the outermost branches will be denoted as the ***leaves***. Only one type of NTAN message is used in the basic protocol, the ***distance update (UPD) message*** (*MSG* in the terminology of [MERL79]). This carries an estimate of the distance to the sink from the sending node.

An ***update cycle*** is initiated by the sink sending a UPD message to all its links with, of course, a distance of zero. When a node receives a UPD message, it adjusts the distance to allow for the added delay of the link and records that distance. If the message arrived along the routing link, the node transmits the shortest distance it has received, by that time in the update cycle, to all the links except the routing link. UPD messages, therefore, propagate from the sink up-tree towards the leaves. Only when a node receives UPD messages from all its links does it send a UPD message back down the routing channel. Of course, this first occurs in the leaves. At this time, the node can select a new routing link if it has found one with a distance shorter than that of the current routing link. The second phase of the update cycle, therefore, consists of UPD messages propagated back down-tree towards the sink. When the sink has received a UPD message back from every link, the update cycle is complete.

In order to extend the basic protocol to cover line failures and other

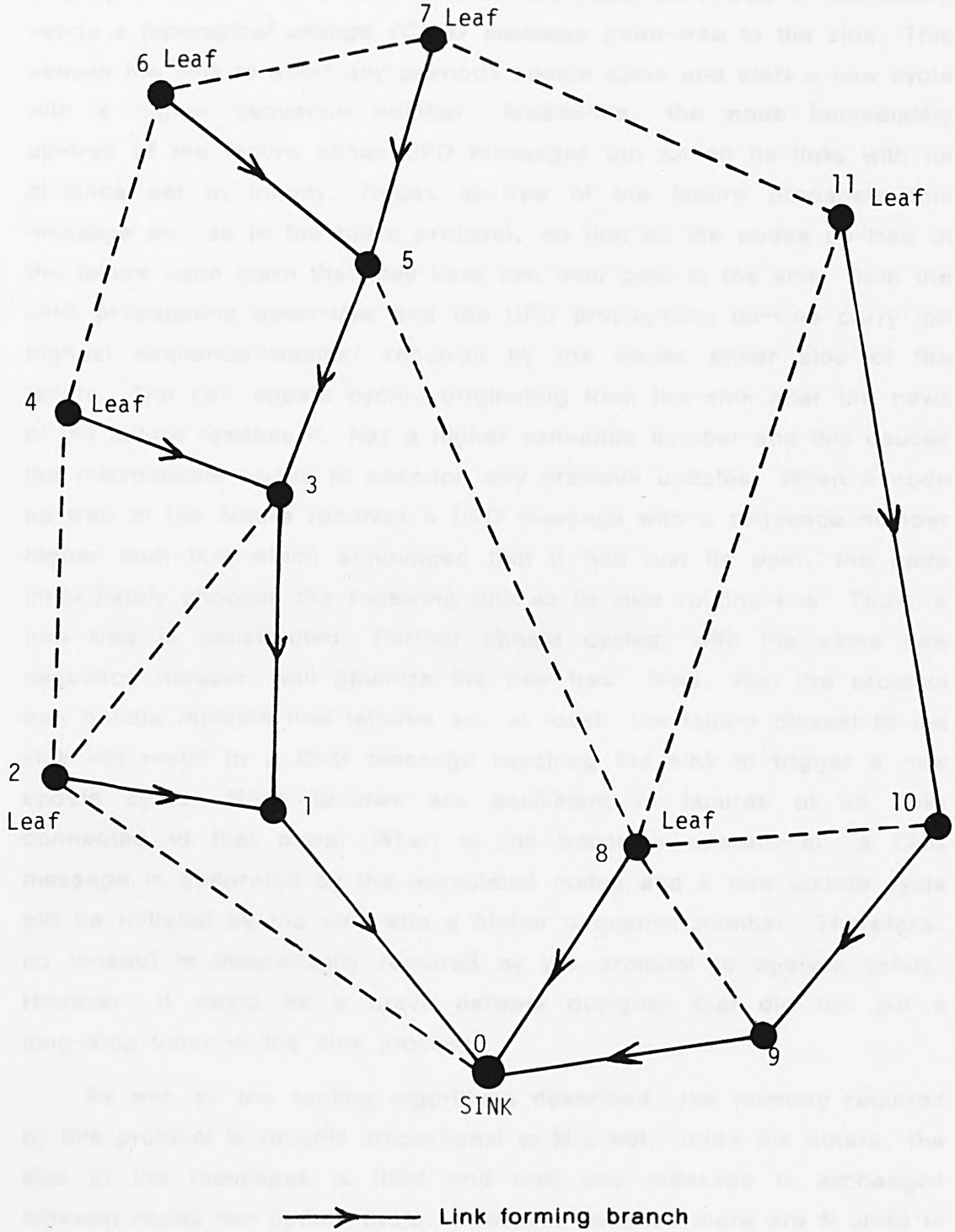


Figure 4.2: A Typical Network Tree

topological changes, the UPD message is lengthened to include a sequence number. When a link fails, the node down-tree of the failure sends a ***topological change (CHG) message*** down-tree to the sink. This causes the sink to abort any previous update cycle and start a new cycle with a higher sequence number. Meanwhile, the node immediately up-tree of the failure sends UPD messages out on all its links with its distance set to infinity. Nodes up-tree of the failure propagate this message on, as in the basic protocol, so that all the nodes up-tree of the failure soon learn that they have lost their path to the sink. Both the CHG propagating down-tree and the UPD propagating up-tree carry the highest sequence number received by the nodes either side of the failure. The new update cycle, originating from the sink after the news of the failure reaches it, has a higher sequence number and this causes the intermediate nodes to abandon any previous updates. When a node up-tree of the failure receives a UPD message with a sequence number higher than that which announced that it had lost its path, the node immediately chooses the receiving link as its new routing link. Thus, a new tree is constructed. Further update cycles, with the same new sequence number, will optimize the new tree. Note, that the protocol can handle multiple line failures as, at least, the failure closest to the sink will result in a CHG message reaching the sink to trigger a new update cycle. Node failures are equivalent to failures of all links connected to that node. When a link becomes operational, a CHG message is generated by the associated nodes and a new update cycle will be initiated by the sink with a higher sequence number. Therefore, no timeout is theoretically required by the protocol to operate safely. However, it would be a brave network designer that did not put a long-stop timer in the sink process!

As with all the routing algorithms described, the memory required by this protocol is roughly proportional to N.L but, unlike the others, the size of the messages is fixed and only one message is exchanged between nodes per update cycle. However, because there are N sinks in the network, the number of messages required to provide an update cycle to every sink is proportional to N. This protocol forms the basis of the sequential MININET routing algorithm. It will be referred to as the ***non-sequential protocol.***

4.3 THE SEQUENTIAL ROUTING PROTOCOL

4.3.1 The Basic Protocol

A path change, in the protocols described so far, could very well lead to a sequence error occurring in one or more of the Virtual Connections established to that sink. In order to make a routing change safely, a node must be sure that the old pathway is clear before it makes the change. To achieve this, a new NTAN message type is introduced, the ***flush control (FLS) message***. An update cycle now consists of four phases instead of the two in the non-sequential version. ***Phase 1*** consists of UPD messages moving up-tree, ***phase 2*** of FLS messages moving down-tree, ***phase 3*** - FLS messages moving back up-tree and finally, ***phase 4*** - UPD messages moving down-tree. The basic protocol only requires that the FLS message carries a variable, f, with two values, ***FLU*** and ***NFL***. When travelling down-tree an FLS message containing f=FLU, ***FLS(FLU)***, acts as a ***flush request message*** while, when travelling up-tree, it acts as a permit to make a routing change. The ***no-flush message***, ***FLS(NFL)***, indicates that no flushing is required when travelling down-tree and a refusal to change route when travelling up-tree. This ability to refuse up-tree route changes is utilized during the recovery from link failures (Section 4.3.2).

Phase 1 proceeds, as with the non-sequential protocol, with UPD messages moving up-tree. When a node receives a UPD message along its routing link, it designates that link as the ***down-tree link***. It then sends, on all other links, a distance estimate (UPD) message based on the shortest distance, from itself to the sink, that it has obtained so far. When any node, including the sink, receives a UPD message from a link, other than the routing link, it responds by sending a no-flush message back along the same link to acknowledge receipt of the UPD message. At this stage, the mutual exchange of UPD and FLS messages only occur along ***cross-tree links*** (i.e. links that do not currently form part of the tree). When a node has received FLS messages from all its links except its down tree link (this first occurs in the leaves), it may decide, on the basis of the distance information then available to the node, to make a routing change. To prepare for the change, the node first freezes packet flow down the old pathway and dispatches a flush request message (FLS(FLU)) down-tree. When a node receives a flush request, it must ensure that all packets in that pathway have been

dispatched down-tree, before it passes on the flush request. Of course, this is not done until FLS messages are received from all the other links. If none of the FLS messages contain flush requests (i.e. they are all FLS(NFL)) and the node does not wish to make a routing change, then a no-flush message is sent down-tree. If a node wishes to make a routing change and also receives a flush request from an up-tree link, it must flush itself before freezing the pathway.

The sink waits until it has received an FLS message from all its links. It then starts phase 3 by sending a flush message out on all links, except those that had already transmitted a no-flush message during phases 1 and 2. When a node receives an FLS message from its down-tree link, it transmits an FLS message along all links that have not previously transmitted an FLS(NFL) as an acknowledgement to a UPD message. The contents of the propagated FLS message is FLU if FLS(FLU) had been received from the down-tree link and the node flushed its buffers during phase 2. Otherwise, FLS(NFL) is transmitted. If the node had frozen the packets flowing down its pathway during phase 2, the receipt of a flush message from its routing link acts as an indication that the old pathway has been successfully flushed. However, the node cannot immediately change its routing link because the new link, with the shortest distance to the sink, may well be connected to a node which is still using the same link in the opposite direction.

Once a node has received a UPD and an FLS message from all links, it sends a UPD message back along the down-tree link. If the node is ready to make a routing change, it selects the link with the shortest distance estimate as its new routing link and releases the sink traffic along this link. This first occurs in the leaves, and then propagates down-tree, until the cycle is completed when the sink has received UPD messages along all its links. Note that, by the completion of the update cycle, a UPD message and an FLS message have been transmitted in both directions along every operational link in the network.

4.3.2 Recovery From Link Failure

If a link which forms a branch fails, the nodes up-tree of the failure cannot directly flush their old paths. This raises three problems that could give rise to sequence errors. Firstly, there is a danger of packets down-tree of the failure arriving at the sink after packets

travelling along a new pathway. Secondly, a diversion pathway may well involve packets having to flow back up-tree. Finally, if more than one diversion pathway is established from different parts of the isolated section, there is a danger that the separate diversions could deliver packets belonging to the same Virtual Connection in the wrong sequence.

As with the non-sequential protocol, the UPD and FLS messages are extended to include a sequence number and the CHG message is introduced. The latter operates in the same manner as in the non-sequential protocol. Also, the vocabulary of the FLS message is extended to include the values DIV and FRT to be described later.

An update cycle follows the same quad-phasic pattern of the basic sequential protocol. However, in addition, during phases 1 and 2, a ***diversion path*** is established (if possible) from the node immediately above the failure to the sink. Also, nodes below the failure are flushed during phase 2. In phases 3 and 4, stranded packets up-tree of the failure are extracted in the correct order along the diversion path and a new tree is established which, once more, spans the entire network. The set of nodes which have lost their path to the sink will be referred to as the ***dead bough*** and the nodes which still have a pathway to the sink will be described as ***live***. The node, immediately above the failure, is the root of the tree formed by the dead bough and will, therefore, be termed the ***failure root***.

When a link forming part of the tree fails (for example the link between nodes 2 and 3 in Figure 4.3), the node immediately up-tree sends a UPD message containing its current sequence number and a distance of infinity. Nodes up-tree of the failure (nodes 3-8 in Figure 4.3) receiving this message freeze traffic destined for the sink. Meanwhile, the node down-tree of the failure (node 2 in Figure 4.3) sends a CHG message, containing the highest sequence number received by that node, back to the sink, indicating that a link has failed. Any intermediate node carrying this message (e.g. node 1 in Figure 4.3) will abort any previously uncompleted update cycle. This message triggers a new cycle with a higher sequence number. When a UPD message of this new cycle (i.e. with a higher sequence number) reaches a live node along its routing link, the node aborts any ongoing update and deletes all previous distance information. It then transmits a revised UPD message on all links except its routing (down-tree) link. If

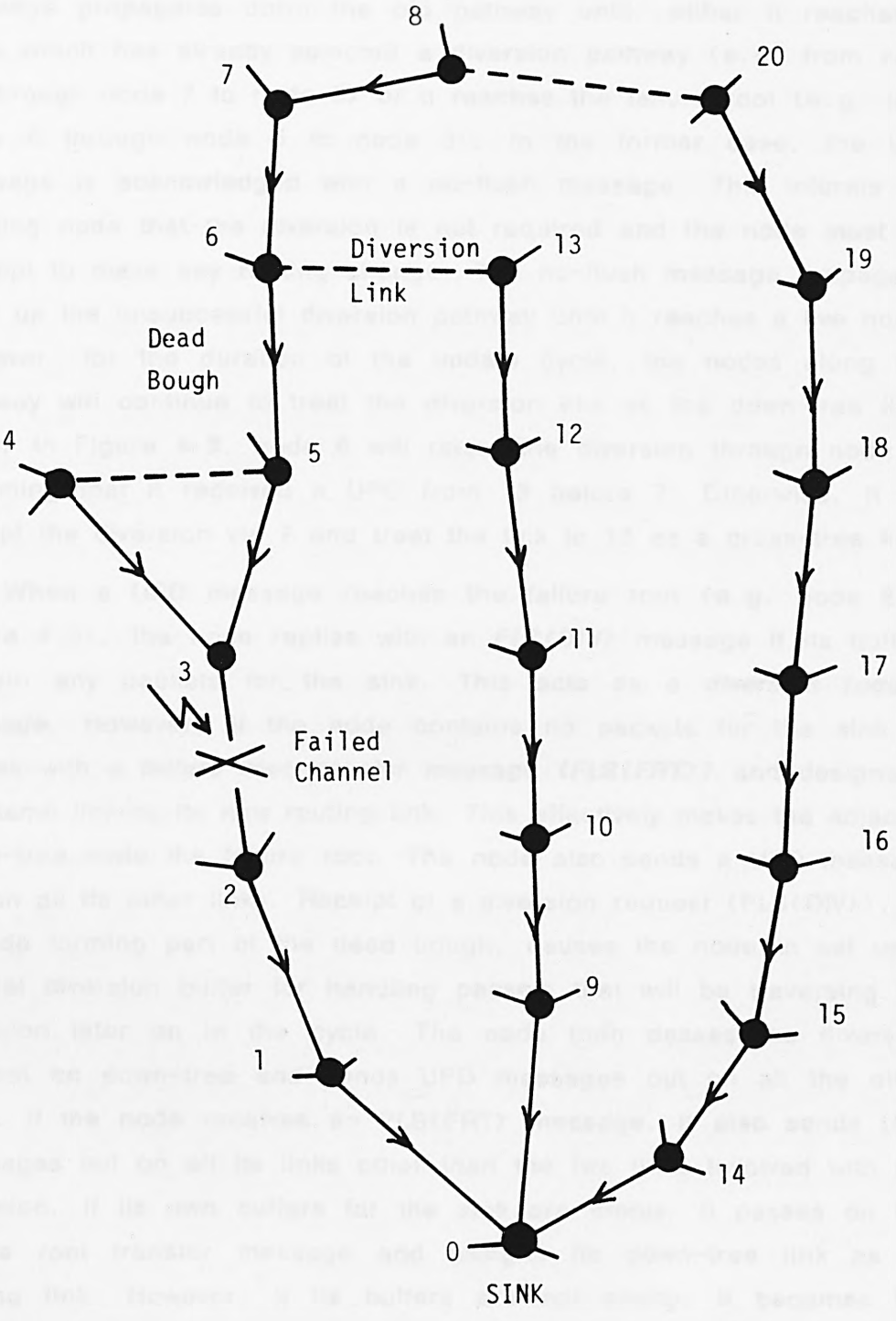


Figure 4.3: Handling Link Failures

an alternative pathway exists around the failure, eventually a UPD will reach a node in the dead bough (e.g. nodes 6 and 8 in Figure 4.3). The node will record the link as its down-tree link and will send a UPD along the old routing link only. Subsequent receipts of any UPD message from other links will result in the node treating them as normal cross-tree links by replying with a no-flush message. Thus, a UPD

message propagates down the old pathway until, either it reaches a node which has already selected a diversion pathway (e.g. from node 8, through node 7 to node 6) or it reaches the failure root (e.g. from node 6 through node 5 to node 3). In the former case, the UPD message is acknowledged with a no-flush message. This informs the sending node that the diversion is not required and the node **must not** attempt to make any routing change. The no-flush message propagates back up the unsuccessful diversion pathway until it reaches a live node. However, for the duration of the update cycle, the nodes along this pathway will continue to treat the diversion link as the down-tree link. Thus, in Figure 4.3, node 6 will reject the diversion through node 7, assuming that it received a UPD from 13 before 7. Otherwise, it will accept the diversion via 7 and treat the link to 13 as a cross-tree link.

When a UPD message reaches the failure root (e.g. node 6 in Figure 4.3), the node replies with an ***FLS(DIV)*** message if its buffers contain any packets for the sink. This acts as a ***diversion request message.*** However, if the node contains no packets for the sink, it replies with a ***failure root transfer message (FLS(FRT))*** and designates the same link as its new routing link. This effectively makes the adjacent down-tree node the failure root. The node also sends a UPD message out on all its other links. Receipt of a diversion request (FLS(DIV)), by a node forming part of the dead bough, causes the node to set up a special diversion buffer for handling packets that will be traversing the diversion later on in the cycle. The node then passes the diversion request on down-tree and sends UPD messages out on all the other links. If the node receives an FLS(FRT) message, it also sends UPD messages out on all its links other than the two links involved with the diversion. If its own buffers for the sink are empty, it passes on the failure root transfer message and assigns its down-tree link as its routing link. However, if its buffers are not empty, it becomes the failure root and sends a diversion request down-tree.

When an FLS(DIV) or FLS(FRT) message eventually reaches a live node, the request is treated as a no-flush message. Thus, in Figure 4.3, assuming that the buffers of node 3 are not empty, nodes 5 and 6 allocate diversion buffers for a diversion path through to node 13. Note, that these diversion buffers are quite distinct from the normal buffer holding packets for the sink. The latter remain frozen.

When nodes in the dead bough, other than those forming the

diversion path, receive a UPD message from their old routing link, they propagate a UPD message out on all their other links except their down-tree link. Those that have not previously received a UPD message of the new update cycle choose the old routing link as their down-tree link. Those that have previously received a UPD message from another link will have already selected that link as their down-tree link and sent a UPD message down the old routing link.

Flush requests and no-flush messages are handled in the same way as in the basic protocol by nodes which have not lost their path to the sink. The node, immediately below the failure, will flush itself during phase 2 and propagate a flush request down-tree. Thus, by the end of phase 2, when the sink has received an FLS message from all links, the path below the failure is flushed, and a diversion has been established between the node immediately above the failure and the sink. The protocol guarantees that the diversion consists of possibly a number of former up-tree links in the dead bough followed by one, and only one, cross-tree link and a number of down-tree links in the live part of the network. Note, that restricting UPD messages to the old routing channel in the dead bough, during phase 1, avoids the danger of the diversion path taking on an obscure form such as via node 4 in Figure 4.3.

Above the diversion path in the dead bough, phases 1 and 2 may still be continuing because the diversion request and any diversion refusals, in the shape of no-flush messages, are propagated down-tree by nodes in the dead bough without waiting for FLS messages to arrive from all the other links. Furthermore, the tree, formed in the update cycle by the set of down-tree links, may well be different to the old routing tree. This is not important, as the protocol forces these nodes not to make any routing changes in the current cycle. At the end of the cycle, the routing link becomes the down-tree link.

Phase 3 is started by the sink sending flush messages out on all links. Nodes in the live section propagate this message on all links that have not earlier transmitted an FLS message. When an FLS message eventually reaches a node in the dead bough along its down-tree link, the node transmits a no-flush message on all other links that have not transmitted an FLS message earlier as a UPD acknowledgement. If the node had received a diversion request from its old routing link (i.e. the node forms part of the diversion), an FLS message is passed on down the diversion path until it reaches the failure root. This informs the node

that the old pathway is flushed and a diversion path has been prepared. It then selects the down-tree link as its routing link for the packets destined for the sink which had been frozen since the old path failed. It also dispatches no-flush messages on all links that have not already transmitted an FLS message in response to a UPD message. The no-flush messages, dispatched on the links not involved in the diversion, serve to block any attempt at a routing change by nodes up-tree from the diversion. When the buffers of the failure root are flushed, it sends a failure root transfer message down-tree. Finally, when it has received UPD and FLS messages from all its operational links, the node transmits a UPD message down-tree. As far as that node is concerned, the update cycle is complete.

Receipt of the root transfer message, by the node immediately down-tree along the diversion path (e.g. node 5 in Figure 4.3), informs the node that the preceding node has flushed the diversion path. Once its diversion buffer is empty, the node can select the diversion link as its routing link and flush its hitherto frozen buffers along the diversion. Thus, the packets, held by that node, will follow the packets that had been trapped in nodes that were down-tree prior to the link failure, so maintaining sequency. When the node is flushed, it sends a failure root transfer message down-tree and, when it has received UPD and FLS messages from all its operational links, the node transmits a UPD message down-tree to complete the update process. The failure root transfer and flushing of the diversion pathway continues down the new tree until it reaches a live node (node 13 in Figure 4.3).

Meanwhile, in other parts of the network, phases 3 and 4 operate as in the basic protocol with nodes propagating flush messages up-tree. Subsequently, having received UPD messages from all links, they return a UPD message down-tree. When the sink has received UPD messages from all its links, the update cycle is complete. Subsequent normal update cycles, with the same sequence number, will optimize the new tree.

4.3.3 Link Recovery

At first sight, the recovery of a channel or node should present very few problems, as there is no loss of communication with the sink along the routing link and the newly restored link can be incorporated by

means of the normal update cycles which will find the new optimum tree. This is true providing the link becomes operational **between** update cycles. However, if the link becomes operational half way through a cycle, it must be excluded from the protocol for that cycle, otherwise lock-out could occur. For example, a node could be waiting for an FLS message from a link that never had a UPD message transmitted earlier. For this reason, each node for each sink maintains a status variable for each link. This variable has the values, DOWN, READY and UP. Normally an operational link has the status UP. If it fails, its status becomes DOWN. If the link recovers during an update cycle, it does not immediately go to the UP state, instead it becomes READY until the cycle is complete whereupon it becomes UP.

The protocol must safeguard against the possibility of one node deciding that a link is UP while the node at the other end leaves it in READY state. This could occur if a new update cycle had already started in one node, but not the other, when the link becomes READY. The non-sequential protocol used a local exchange of messages to attempt to synchronize the rehabilitation of the link [MERL79]. The link is not accepted as UP until the start of a new cycle with a higher sequence number than that received hitherto by either node. Since agreement in a distributed protocol can never be reached simultaneously, any deadlock situation, which could arise, is resolved by the new update cycle. The new cycle is triggered by the dispatch, down tree, of a CHG message bearing the highest sequence number received by the node. Receipt of the messages causes the sink to abort any ongoing update cycle, just as in the case of a link failure.

This approach is not so attractive for the sequential protocol, basically because here, a path change is more costly, as user traffic is temporarily blocked while the old pathway is flushed. Thus, it is undesirable to abort an update cycle, even though the new cycle, including the recovered link, might result in an improved tree. It is much better to complete the current update cycle and get packets moving again, and then to optimize the tree by starting the new cycle with a higher sequence number. Of course, a link failure must cause the immediate abortion of any current update cycle and the start of a new cycle with a higher sequence number. The CHG message is extended to include a binary variable, which may have the values ***FAIL*** or ***REC*** to distinguish between link failures and recoveries respectively.

Instead of attempting to block a link becoming UP one side and only READY the other, this protocol attempts to avoid deadlock if and when link asymmetry occurs. A node will automatically make a READY link UP if it is not currently performing an update cycle. If, however, it receives a UPD message from a link which is not UP, after it has broadcast UPD messages in phase 1 of the cycle, but before it has performed phase 2, the node assigns the status UP to the link and sends a UPD message down the link. In other words, the link is allowed to catch up with the other links. If, however, the node has already sent an FLS message down-tree the link status does not change. Nevertheless, the node transmits an FLS(NFL) and a UPD message down the link so as to satisfy the protocol requirements of the adjacent node. The link will eventually be accepted by the node because a CHG message will result in a new update cycle with a higher sequence number.

Because of the link hold-down process practised by the channel managers at each end of the link (Section 2.3.3), one side could decide that the link had recovered while the other was still treating it as down. This means that the protocol may receive a UPD message along a link which it considers to be DOWN.

4.3.4 Multiple Failures

The protocol, so far described, will successfully handle two or more link failures provided that sufficient time occurs between each failure for an update cycle to repair the damage. In addition, the protocol performs satisfactorily if a number of disjoint failures - i.e. failures of links carrying no common pathway - or if a number of contiguous failures - i.e. failures of nodes and links forming a contiguous section of a pathway - occur simultaneously. The most likely cause of these types of failure is that of a single node as shown in Figure 4.4.

There are three ways, however, in which a second failure, occurring anywhere in the network very soon after the first failure, can cause problems.

(1) The second failure occurs down-tree from the first failure after the CHG(FAIL) message had been transmitted down-tree but before a new cycle could flush the old pathway. This is illustrated in

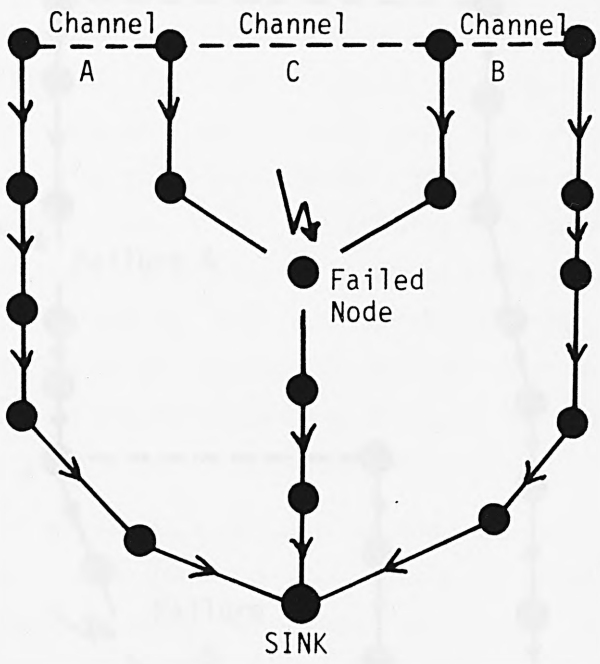


Figure 4.4: Node Failure

Figure 4.5 assuming that Failure A occurred before Failure B.

(2) The second failure occurs up-tree of the first failure before a diversion could be established and flushed. This is shown in Figure 4.5 if Failure A occurred after Failure B.

(3) The second failure causes the update cycle to restart during the time that the diversion around the first failure is being flushed. The nodes along the diversion pathway cannot abort the flushing operation without the danger of sequence errors.

The first two cases result in some pathways being split into three parts. The protocol, so far described, would set up two diversions operating simultaneously, which would result in the distinct possibility of sequence errors.

The approach to these problems, adopted in this protocol, is based on the fact that the time taken to perform an update cycle (probably a few milliseconds in MININET) is infinitesimal compared with the mean time between separate channel failures. Therefore, these are extremely unlikely events. Consequently, as packet loss is more acceptable than loss of sequency in MININET (Section 1.2.4), the aim of the protocol, when faced with multiple failures, is to recover without loss of sequency but allowing packets to be dropped. The basic method is firstly to detect that a multiple failure has occurred, and then to drop packets if there is

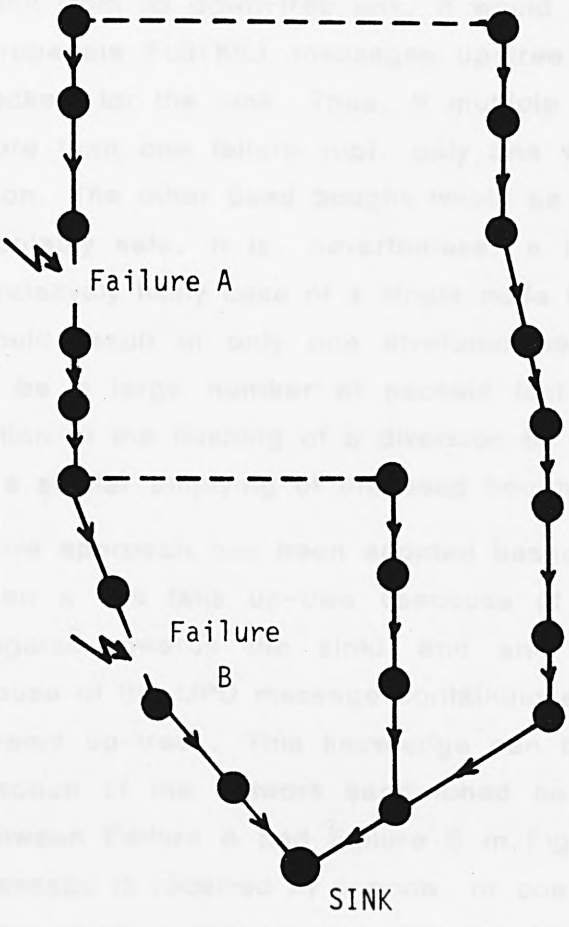


Figure 4.5: Non-Contiguous Failures Along a Common Pathway

any danger of sequence errors.

In order to be able to propagate the detection of a multiple failure within a dead bough, the vocabulary of the FLS message is extended, yet again, to include the value ***KIL***, which causes the up-tree receiving node to drop all packets for the sink. In addition, the node will propagate the message on up the routing tree.

There are a number of methods which could be used to recover from multiple failures. One, initially very attractive, is to allow only one diversion to be set up in any one update cycle. The diversion request would be relayed all the way to the sink during phase 2, with the intermediate nodes recording the link, from which they received the request, as their diversion link. During phase 3, the sink would transmit not more than one diversion permit and this is relayed back up the diversion links until it reaches the dead bough. If there is more than one failure root, only one would receive the permit. If a node, which forms part of the diversion, receives an FLS message which does not contain

a diversion permit from its down-tree link, it would drop all packets for the root and propagate FLS(KIL) messages up-tree to empty the dead bough of all packets for the sink. Thus, if multiple failures lead to the existence of more than one failure root, only one would be allowed to set up a diversion. The other dead boughs would be emptied. While this solution is undeniably safe, it is, nevertheless, a little too drastic. In particular, the relatively likely case of a single node failure, as shown in Figure 4.4, would result in only one diversion being permitted while there may well be a large number of packets lost in the other dead bough. Interruption of the flushing of a diversion by a new update cycle would result in a similar emptying of the dead bough.

An alternative approach has been adopted based on the fact that a node knows when a link fails up-tree (because of the CHG message which is propagated towards the sink) and also when a link fails down-tree (because of the UPD message containing a distance of infinity which is propagated up-tree). This knowledge can be used to drop all packets in a section of the network sandwiched between two failures, such as that between Failure A and Failure B in Figure 4.5. Whenever a CHG(FAIL) message is received by a node, or one of its up-tree links fails, a flag is set which is only cleared after the node reaches phase 3 of a new update cycle. If the node is live - or thinks it is - the CHG(FAIL) message causes the node to abort any ongoing update cycle and await a new cycle with a higher sequence number. If the node then receives a UPD message from its routing link, which indicates that it has lost its path to the sink, and this flag is still set, the node drops all packets addressed to the sink. This handles the first type of multiple failure described earlier. If a node within a dead bough receives a CHG(FAIL) message or an up-tree link fails, then it also drops all packets for the sink. Furthermore, if the node is actively involved with the diversion operation it propagates FLS(KIL) messages to all up-tree links. This is done because, while the diversion is being flushed, nodes up-tree of the diversion will have received normal UPD and FLS messages up-tree and therefore would not be aware that they still form part of the dead bough. The FLS(KIL) messages ensure that the entire bough is emptied. This may appear rather drastic, but remember that this type of failure is an extremely unlikely situation. This procedure protects against the second type of multiple failure. The third type of multiple failure can now only occur if the second failure, which triggered

the new update cycles, is not up-tree of the original failure - otherwise the diversion would have been aborted when the CHG(FAIL) message was received. It is only necessary to drop packets in nodes along the diversion if a new update cycle is started before the diversion is flushed.

The protocol, therefore, will recover from single channel or node failures without loss of packets and from multiple failures occurring simultaneously with the possible loss of some packets.

If the network becomes partitioned, i.e. no pathway can be found to one or more sinks, then a dead bough remains isolated until repairs are effected. In terms of the network timescale, this could be an unconscionably long time. In the interim, resources are tied up in the dead bough with buffers containing packets that cannot be delivered. Worst still, when the network is reunited, there is the danger that the current sequence number of the live network has incremented more than half circle from its value when the network was partitioned. This would result in the dead node ignoring the new update cycle as it would appear to have a lower sequence number. For these reasons, it is desirable for the nodes in the dead bough to maintain a timer while they are in a dead state. After a certain time without any new update cycle reaching the node, it is reasonable for the node to assume that no path exists to the sink. The node can then drop all packets destined for the sink and release the buffer allocation. It may then forget the existence of the sink. Of course, it is necessary for the nodes up-tree of the failure to also reset at the same time. This is performed by the resetting node transmitting a further extension of the FLS message, ***FLS(RST)***, to all up-tree nodes. FLS(RST) has the same effect as the KIL message but, in addition, causes the node to release any resources permanently allocated to the sink and forget the last sequence number. When a pathway is finally re-established to the sink, the protocol operates normally as during network initialization, with the first UPD message received defining the routing and down-tree links and the current sequence number.

4.4 THE ALGORITHM PERFORMED BY THE NODES

A separate copy of the algorithm operates for each sink in the network. Since each algorithm is distributed across every node in the network, each node must contain a separate routing process for each

sink (except itself) plus its own sink process. Thus, each routing process must be identified by the node, n, where it resides and the sink, r, to which it pertains. However, the following description of the operation of these processes will exclude indexing all variables with n and r for the sake of brevity. Similarly all messages both inter-nodal and intra-nodal must contain the sink identification, r, which will also be suppressed in the following description.

The routing process in each node interacts, not only with routing processes in adjacent nodes by means of NTAN inter-node messages, but also with the channel manager and buffer control processes within the same node. This is performed by means of intra-node messages as shown in Figure 4.6. The types of messages exchanged are described in Table 4.1.

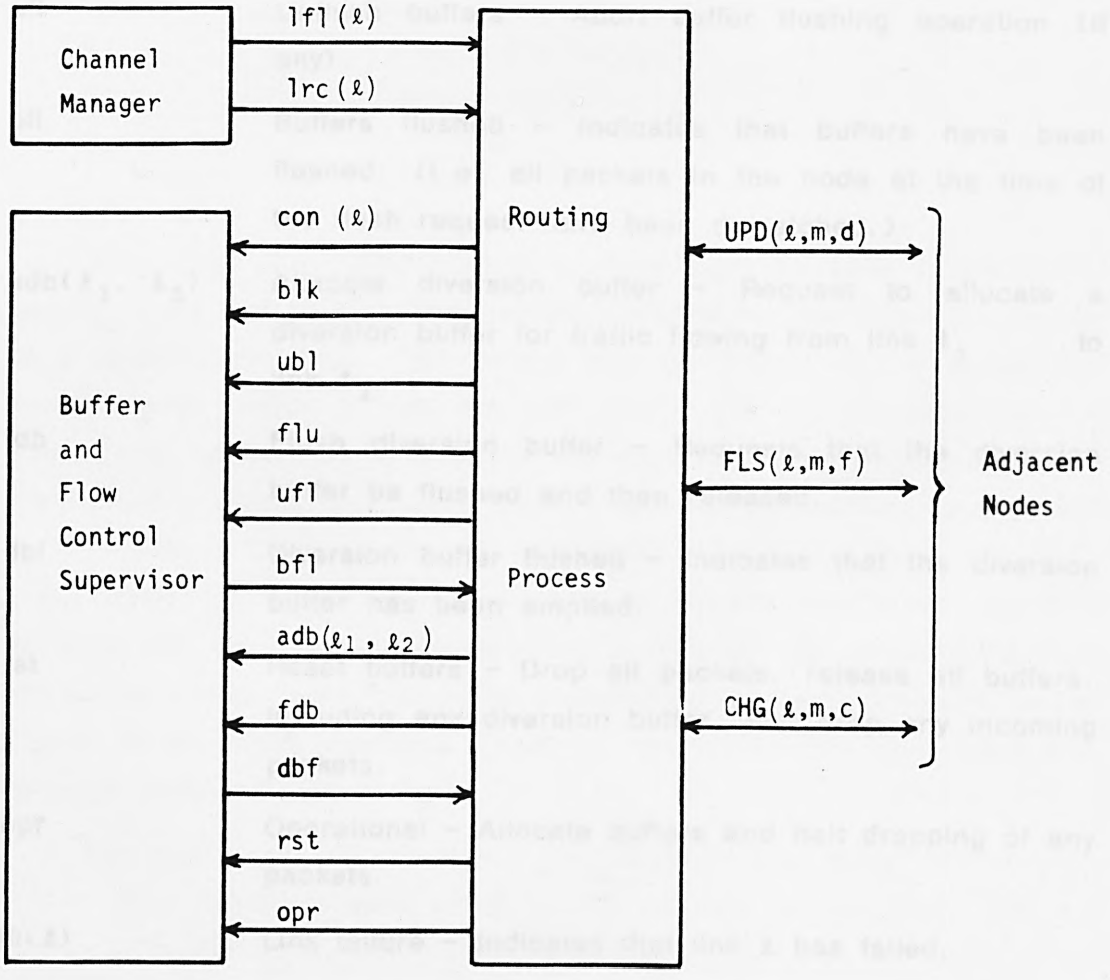


Figure 4.6: Intra-Node and Inter-Node Message Transfers

Table 4.1

Routing Algorithm Message Types

Note that all messages (except lfl and lrc), buffers, traffic, etc. refer **only** to a specific sink, r.

I Intra-node Messages

con(ℓ)	Connect buffers - Directs all output from the packet buffers to link ℓ. Note that the buffers can be frozen by the message, con(nil).
blk	Block input - Block all traffic input to the node.
ubl	Unblock input - Allow packets into the node. (i.e. allow normal flow control mechanisms to operate.)
flu	Flush buffers - Requests buffer to be flushed.
ufl	Unflush buffers - Abort buffer flushing operation (if any).
bfl	Buffers flushed - Indicates that buffers have been flushed. (i.e. all packets in the node at the time of the flush request have been dispatched.)
adb(ℓ_1, ℓ_2)	Allocate diversion buffer - Request to allocate a diversion buffer for traffic flowing from link ℓ_1 to link ℓ_2.
fdb	Flush diversion buffer - Requests that the diversion buffer be flushed and then released.
dbf	Diversion buffer flushed - Indicates that the diversion buffer has been emptied.
rst	Reset buffers - Drop all packets, release all buffers, including any diversion buffer, and drop any incoming packets.
opr	Operational - Allocate buffers and halt dropping of any packets.
lfl(ℓ)	Link failure - Indicates that link ℓ has failed.
lrc(ℓ)	Link recovery - Indicates that link ℓ is once again operational.

II Inter-Node Messages

UPD(ℓ, m, d) Distance update message sent along link ℓ, containing sequence number m, and distance to sink d.

FLS(ℓ, m, f) Flush control message sent along link ℓ, containing sequence number m, and flush control variable f which may have the values:

NFL = no-operation;
FLU = flush request/grant;
DIV = diversion link request;
FRT = failure root transfer;
KIL = kill buffers and reset;
RST = kill buffers and forget the sink.

CHG(ℓ, m, c) Change message sent along link ℓ, containing sequence number m, and of type c which may have the values:

FAIL = link failure;
REC = link recovery.

Note, that the buffer control process must be able to freeze the buffers, i.e. halt all output of packets destined for the sink upon receipt of a ***con(nil)*** message. The buffers can be thawed with a subsequent ***con*** message which defines the new output link for all traffic for the sink. Thus, actual routing changes are made by freezing the buffers and then thawing with a new output link. The blocking operation stops input to the node. A blocking request, ***blk***, does not have to come into effect immediately. However, it is expected to be effective within the time required for the flow control process to request the adjacent up-tree nodes to freeze their output by transmitting a back pressure vector (BPV) in an NTAN message. That is, within the BPV turnround time between adjacent nodes.

The flush request message, ***flu***, requires the buffer manager to inform the routing process when all the packets for the sink, that are currently in the node, are safely transmitted. This may be implemented in number of ways. If the packets or buffer slots are distinguishable, the manager can simply tag the packets currently in the node and wait until all the tagged packets have been transmitted. Similarly, if the internal

buffer organization is a strict first-in, first-out queue, then the manager need only note the number of packets present when the request is made and wait until that number has been transmitted. However, if the queue structure is more complex and tagging is impractical, as may well be the case in MININET, then a sure-fire method of flushing the node is to block inputs and wait until the node empties, whereupon the inputs are unblocked and the routing process is informed. Unfortunately, this does cause some undesirable perturbation of the traffic flow.

In order to provide a diversion pathway, the node must not only provide a diversion queue allocation attached to the down-tree link, but also selectively unblock only the diversion input link leaving the other links blocked. The reset command, ***rst***, is used, after multiple errors or after a long period with all pathways to the sink lost, to drop all sink packets, both in the normal queue, and in the diversion buffer. When the node initially receives a UPD message from the sink and during recovery from multiple failures, the operational message, ***opr***, is used to set up the buffer handling control mechanisms for packets destined for the sink.

Following Merlin and Segall [MERL79], the routing algorithm can be expressed as a set of FSMs operating in each node. The FSMs of all nodes other than the sink are identical. Each FSM is driven by a message handler which receives both inter-node and intra-node messages.

4.4.1 The Sink Algorithm

The state diagram for the sink is shown in Figure 4.7. Tables 4.2, 4.3 and 4.4 define the variables used, the message handler algorithm and the FSM transition algorithm respectively. During phases 1 and 2 of an update cycle, the sink is in state S1. When it has received FLS messages from all links that are UP, it moves to state S2 where it normally stays until the end of the update cycle. Before starting a new cycle, the node waits a fixed time in state SW. This is because, unlike the non-sequential algorithm, update cycles can interrupt user traffic while old pathways are being flushed. This loss of user throughput is in addition to the usual loss of effective channel capacity due to the routing messages. It will be shown in Section 4.5 that, after two cycles with no-flush requests reaching the sink, the tree is optimized and the sink

Table 4.2

Variables Used in Each Sink Process

S	Major state, ϵ {SI, S1, S2, SW, SQ}
Mc	Current sequence number, ϵ {0, 1, 2... Mmax}
N	Cycle counter, ϵ {0, 1, 2}
H	CHG received flag ϵ {0, 1}

For each link, ℓ = 1, 2... Lmax:

C(ℓ)	Link status, ϵ {DOWN, READY, UP}
U(ℓ)	UPD received flag, ϵ {0, 1}
F(ℓ)	FLS received flag, ϵ {0, 1}

Table 4.3

Sink Message Handler Algorithm

```
For UPD(ℓ, m = Mc, d):
    if C(ℓ) ≠ UP then
        ( if S = S1 then ( C(ℓ) ← UP; send UPD(ℓ, Mc, 0) );
          if S = S2 then
             ( send UPD(ℓ, Mc, 0);
               send FLS (ℓ, Mc, NFL) ) );
    if S = S1 then send FLS (ℓ, Mc, NFL);
    U(ℓ) ← 1; execute FSM.

For FLS(ℓ, m = Mc, f):
    F(ℓ) ← 1;
    if C(ℓ) = UP then
        ( if f = FLU then N ← 0; execute FSM ).

For CHG(ℓ, m = Mc, c):
    H ← 1; execute FSM.

For lfl(ℓ):
    H ← 1; C(ℓ) ← DOWN; execute FSM.

For lrc(ℓ):
    H ← 1; if C(ℓ) = DOWN then C(ℓ) ← READY;
    execute FSM.
```

Table 4.4

State Transitions in the Sink Process

init → S1	Condition:	node initialization.
	Action:	Mc ← 0; N ← 0; H ← 0; start timer T_A;
		∀ i then (U(i) ← 0; F(i) ← 0;
		if link i operational then C(i) ← UP
		else C(i) ← DOWN);
		∀ i s.t. C(i) = UP, send UPD(i, Mc, 0).
S1 → S2	Condition:	∀ i, s.t. C(i) = UP then F(i) = 1.
	Action:	∀ i s.t. (C(i) = UP)∧(U(i) = 0) then
		send FLS(i, Mc, FLU).
S2 → SW	Condition:	∀ i, s.t. C(i) = UP, (U(i) = 1)∧(N < 2).
	Action:	Start timer T_W.
S2 → SQ	Condition:	∀ i s.t. C(i) = UP, (U(i) = 1)∧(N ≥ 2).
	Action:	None.
S1 → S1	Condition:	CHG (ℓ, m, c = FAIL) ∨ lfl(ℓ)
		∨ time T_A expired.
SQ → S1	Condition:	H = 1.
SW	Action:	∀ i s.t. C(i) = READY, C(i) ← UP;
		Mc ← Mc + 1; N ← 1; H ← 0; start timer T_A;
		∀ i, (U(i) ← 0; F(i) ← 0);
		∀ i s.t. C(i) = UP, send UPD (i, Mc, 0).
SW → S1	Condition:	time period T_W expired.
	Action:	∀ i s.t. C(i) = READY, C(i) ← UP;
		N ← N + 1; H ← 0; start timer T_A;
		∀ i, (U(i) ← 0; F(i) ← 0);
		∀ i s.t. C(i) = UP, send UPD (i, Mc, 0);

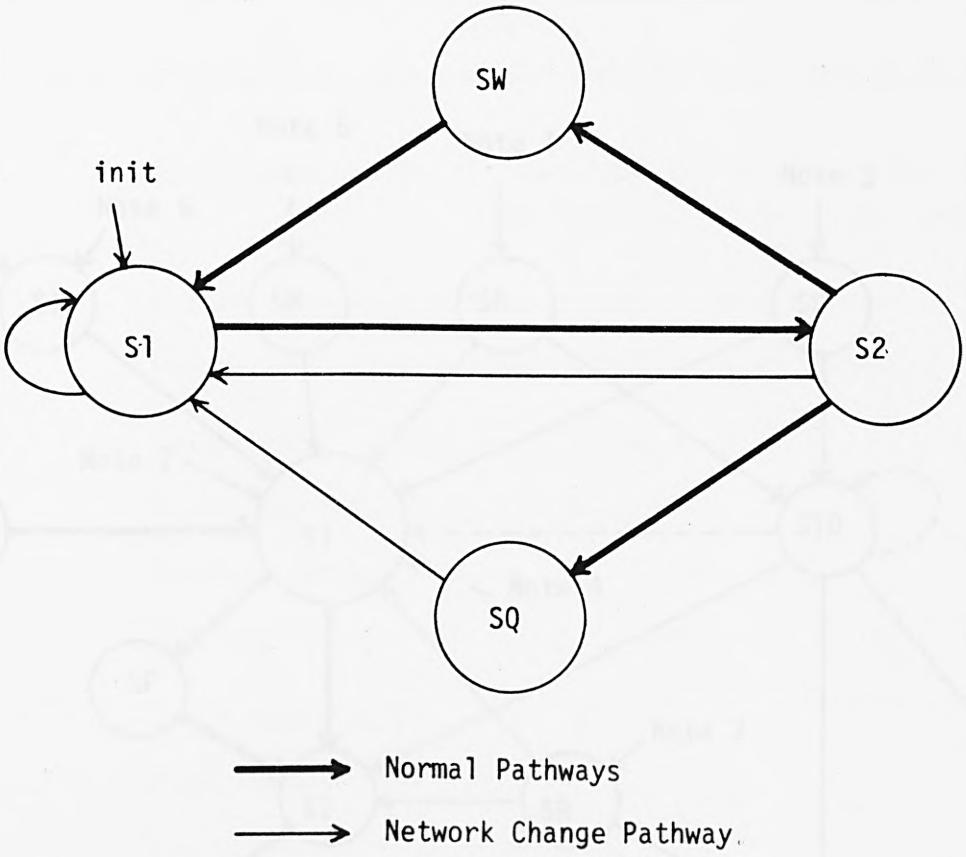


Figure 4.7: State Diagram of the Routing Process in the Sink

enters the quiescent state, SQ. It only leaves this state when a CHG message is received or one of its links changes status.

If, at any time in the cycle, the sink receives a CHG(FAIL) message or one of its own links fails or the "dead-man's timer" of period T_A times-out, then the process immediately starts a new cycle with an incremented sequence number and enters S1. The dead-man's timer is a "long-stop" timer to guard against (the theoretically impossible) deadlock.

4.4.2 The Intermediate Node Algorithm

Figure 4.8 shows the state diagram of the routing process in an intermediate node. The variables used in this process are listed in Table 4.5. Its message handler and FSM algorithms are described in Tables 4.6 and 4.7 respectively. The initialization state, SI, is included for convenience. In a practical implementation, where the process may well not be created until the existence of the sink is disclosed by the arrival of a UPD message, the state SI would not be necessary.

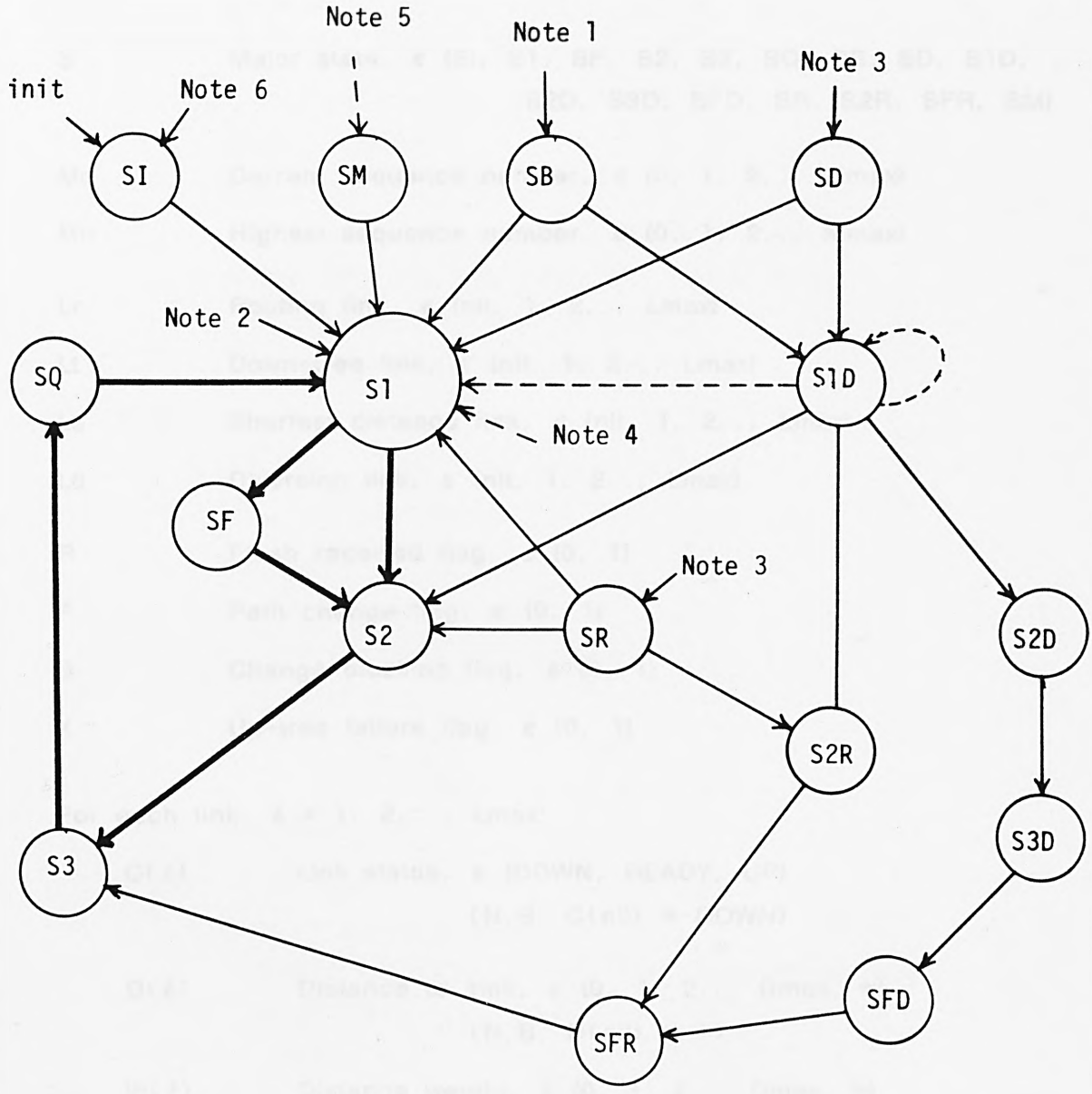


Figure 4.8: State Diagram of the Intermediate Node Routing Process

Table 4.5

Variables Used in an Intermediate Node for Each Sink Process

S	Major state, ϵ {SI, S1, SF, S2, S3, SQ, SB, SD, S1D, S2D, S3D, SFD, SR, S2R, SFR, SM}
Mc	Current sequence number, ϵ {0, 1, 2... Mmax}
Mh	Highest sequence number, ϵ {0, 1, 2... Mmax}
Lr	Routing link, ϵ {nil, 1, 2... Lmax}
Lt	Down-tree link, ϵ {nil, 1, 2... Lmax}
Ls	Shortest distance link, ϵ {nil, 1, 2... Lmax}
Ld	Diversion link, ϵ {nil, 1, 2... Lmax}
R	Flush received flag, ϵ {0, 1}
P	Path change flag, ϵ {0, 1}
B	Change blocking flag, ϵ {0, 1}
X	Up-tree failure flag, ϵ {0, 1}

For each link, ℓ = 1, 2,... Lmax:

C(ℓ)	Link status, ϵ {DOWN, READY, UP} (N.B. C(nil) $\equiv$ DOWN)
D(ℓ)	Distance to sink, ϵ {0, 1, 2... Dmax, ∞} (N.B. D(nil) $\equiv \infty$)
W(ℓ)	Distance weight, ϵ {0, 1, 2... Dmax, ∞}
U(ℓ)	UPD received flag, ϵ {0, 1}
F(ℓ)	FLS received flag, ϵ {0, 1}

The states S1, SF, S2, S3 and SQ are the states used in a normal update cycle. Upon receipt of a UPD message from the down-tree link, the process performs phase 1 by propagating UPD messages up-tree and enters S1. When it has received FLS messages from all its up-tree links, the process enters SF if it has received any flush requests (indicated by the flag R). If no flush requests had been received, the node performs phase 2 by sending an FLS message down-tree and

Table 4.6

Message Handler for an Intermediate Node Routing Process

```
For UPD(ℓ, m ⩾ Mc, d) ∀ ( UPD(ℓ, m, d)∧(S = SI) ):
    if (C(ℓ) ≠ UP)∧(m = Mc) then
        ( C(ℓ) ← READY;
          if S ∈ {S1, S2D, S2R} then
             ( C(ℓ) ← UP; send UPD(ℓ, Mc, D(Ls)) );
          if S ∈ {SF, S2, S3, S3D, SFD, SFR} then
             ( send UPD(ℓ, Mc, D(Ls));
               send FLS(ℓ, Mc, NFL) ) );
    if m > Mh then
        ( ∀ i then ( D(i) ← ∞; U(i) ← 0; F(i)
          Mh ← m; Ls ← nil );
    if m = Mh then
        ( d ← d + W(ℓ); U(ℓ) ← 1;
          if (ℓ = Ls)∧(d > D(ℓ)) then
             ( D(ℓ) ← d;
               ∀ i s.t. C(i) = UP then
                     if D(i) < D(Ls) then Ls ← i )
          else D(ℓ) ← d );
    if C(ℓ) = UP then
        ( if (ℓ ≠ Lt)∧(m = Mc)
             ∧(S ∈ {SQ, S1, SF, S2, S1D, S2D, S2R} then
             send FLS(ℓ, Mc, NFL);
          if D(ℓ) < D(Ls) then Ls ← ℓ;
          execute FSM ).

For FLS(ℓ, m = Mh, f):
    F(ℓ) ← 1;
    if C(ℓ) = UP then
        ( if (f = FLU)∧(ℓ ≠ Lt) then R ← 1;
          if (m = Mc) ∨ (f ∈ {KIL, RST}) then execute FSM ).
```

```
For CHG(ℓ, m, c):
    if C(Lr) = UP then send CHG(Lr, m, c)
    else if C(Lt) = UP then send CHG (Lt, m, c);
    if c = FAIL  then ( if ℓ ≠ Lr then X ← 1; execute FSM ).

For lfl(ℓ):
    C(ℓ) ← DOWN; R ← 1; if ℓ ≠ Lr then X ← 1; D(ℓ) ← ∞;
    if C(Lr) = UP then send CHG(Lr, Mc, FAIL)
    else if C(Lt) = UP then send CHG(Lt, Mc, FAIL);
    execute FSM.

For lrc(ℓ):
    if C(ℓ) = DOWN then C(ℓ) ← READY;
    if C(Lr) = UP then send CHG(Lr, Mc, REC)
    else if C(Lt) = UP then send CHG(Lt, Mc, REC);
    if S ∈ {SQ, SB, SD, S1D, SR, SM} then C(ℓ) ← UP ).

For bfl, dbf:
    execute FSM.
```

enters S2. After flushing its buffers, the process leaves SF and enters S2 performing phase 2. It is at this time that the node decides whether to make a routing change. If there is another link having a shorter distance than the current routing link (i.e. D(Ls) < D(Lr)), then the process freezes sink traffic and sets the flag P. Upon receipt of the FLS message from its down-tree link, the node performs phase 3 by sending FLS messages to all those remaining links to which it had not already sent an FLS message. It then enters S3. Finally, when UPD messages have been received from all links, the node makes a routing change if requested and permitted, performs phase 4 by sending a UPD message down-tree and enters the quiescent state, SQ. Note that, if a node is a leaf, it may pass straight through states S1 and S3. Therefore, the implementation cannot assume only one state transition per event.

If the node receives a UPD message from a link, other than Lr, with a higher sequence number than the node's current sequence number, Mc, the process enters the standby state, SB. If, eventually, a UPD message with the same sequence number reaches the node via

Table 4.7

State Transitions in an Intermediate Node Routing Process

init → S1 Condition: node initialization.
Action: none.

SI → S1 Condition: UPD (ℓ, m, d ≠ ∞).
Action: Lr ← ℓ; Lt ← ℓ; Ls ← ℓ; Ld ← nil;
Mc ← m; Mh ← m;
R ← 0; P ← 0; B ← 0; X ← 0;
∀ *i* then
(D(*i*) ← ∞; U(*i*) ← 0; F(*i*) ← 0;
if link *i* operational then C(*i*) ← UP
else C(*i*) ← DOWN);
D(ℓ) ← d; U(ℓ) ← 1;
∀ *i* ≠ Lt s.t. C(*i*) = UP then
send UPD(*i*, Mc, D(Ls));
send opr; send ubl; send con(Lr).

S1 → SF Condition: (∀ *i* ≠ Lt s.t. C(*i*) = UP then F(*i*)) Λ R Λ B'.
Action: send fls.

S1 → S2 Condition: (∀ *i* ≠ Lt s.t. C(*i*) = UP then F(*i*))Λ(R' ∨ B).
Action: if (D(Ls) < D(Lr)) Λ B' then
(send con(nil); P ← 1;
send FLS(Lt, Mc, FLU))
else send FLS (Lt, Mc, NFL).

SF → S2 Condition: bfl.
Action: if D(Ls) < D(Lr) then
(send con(nil); P ← 1);
send FLS(Lt, Mc, FLU).

S2 → S3 Condition: FLS(ℓ = Lt, m, f ≠ KIL).
Action: if P Λ ((f ≠ FLU) ∨ B) then
(send con(Lr); P ← 0);
if B ∨ R' then f ← NFL;
R ← R Λ (f ≠ FLU); X ← 0;
∀ *i* s.t. (C(*i*) = UP) Λ U(*i*)' then
send FLS(*i*, Mc, f).

S3 → SQ Condition: $\forall i$ s.t. C(i) = UP then U(i) Λ F(i).
Action: if P then (Lr ← Ls; send con(Lr); P ← 0);
send UPD (Lt, Mc, D(Lr));
Lt ← Lr;
$\forall i$ s.t. C(i) = READY then C(i) ← UP;
$\forall i$ then (U(i) ← 0; F(i) ← 0).

SQ → S1 Condition: UPD(ℓ = Lr, m, D ≠ ∞).
Action: Mc ← m; B ← 0; Ld ← nil;
$\forall i \neq$ Lt s.t. C(i) = UP then
send UPD(i, Mc, D(Ls)).

S1 → SB Condition: UPD($\ell \neq$ Lr, m > Mc, d) $\vee$ Ifl($\ell \neq$ Lr)
SF $\vee$ CHG(ℓ, m, FAIL).
S2 Action: Lt ← Lr; if UPD then Ls ← ℓ;
S3 if P then (send con(Lr); P ← 0);
SQ send ufl;
SFR $\forall i$ s.t. C(i) = READY then C(i) ← UP.

SB → S1 Condition: UPD (ℓ = Lr, m = Mh, d ≠ ∞).
Action: Mc ← m; B ← 0; Ld ← nil;
$\forall i \neq$ Lr s.t. U(i) Λ (C(i) = UP) then
send FLS(i, Mc, NFL);
$\forall i \neq$ Lt s.t. C(i) = UP then
send UPD (i, Mc, D(Ls)).

S1 → S1 Condition: UPD (ℓ = Lr, m > Mc, d ≠ ∞).
SF Action: Mc ← m; Lt ← Lr; Ld ← nil; B ← 0;
S2 if P then (send con(Lr); P ← 0);
S3 send ufl;
SFR $\forall i$ s.t. C(i) = READY then C(i) ← UP;
$\forall i \neq$ Lt s.t. C(i) = UP then
send UPD(i, Mc, D(Ls)).

S1 → SR Condition: lfl(ℓ = Lr) Λ X'.

SF Action: $\forall$ *i* then (D(*i*) ← ∞; U(*i*) ← 0; F(*i*) ← 0);

S2 send con(nil); send blk; send ufl;

S3 start timer T_D; Ls ← nil; Ld ← nil;

SQ P ← 0; R ← 0;

SB $\forall$ *i* s.t. C(*i*) = READY then C(*i*) ← UP;

SFR $\forall$ *i* s.t. C(*i*) = UP then send UPD(*i*, Mc, ∞).

S1 → SD Condition: UPD (ℓ = Lr, m = Mh, d = ∞) Λ X'

SF Action: $\forall$ *i* then (D(*i*) ← ∞; U(*i*) ← 0; F(← 0);

S2 send con(nil); send blk; send ufl;

S3 start timer T_D; Ls ← nil; Ld ← Lr;

SQ Mc ← m; P ← 0; R ← 0;

SB $\forall$ *i* s.t. C(*i*) = READY then C(*i*) ← UP;

SFR $\forall$ *i* ≠ Lr s.t. C(*i*) = UP then
send UPD(*i*, Mc, ∞).

SD → S1 Condition: UPD (ℓ = Lr, m > Mc, d ≠ ∞).

S1D Action: Mc ← m; Lt ← Lr; Ld ← nil;

SR send con(Lr); send ubl;
$\forall$ *i* ≠ Lt s.t. C(*i*) = UP then
send UPD(*i*, Mc, D(Ls)).

SD → S1D Condition: UPD(ℓ ≠ Lr, m > Mc, d ≠ ∞).

S1D Action: Mc ← m; Lt ← ℓ; send UPD (Ld, Mc, D(Ls));
B ← 1.

SB → S1D Condition: UPD(ℓ = Lr, m < Mh, d = ∞) Λ X'.

Action: send con(nil); send blk; B ← 1; R ← 0;
Mc ← Mh; Lt ← Ls; Ld ← Lr;
$\forall$ *i* ≠ Lr s.t. C(*i*) = UP then
send UPD(*i*, m, ∞);
send UPD(Ld, Mc, D(Ls)).

SR → S2R Condition: UPD(ℓ ≠ Lr, m > Mc, d ≠ ∞)
Λ buffers non-empty.

Action: Mc ← m; Lr ← nil; Lt ← ℓ; B ← 1;
send FLS(Lt, Mc, DIV);
$\forall$ *i* ≠ Lt s.t. C(*i*) = UP then
send UPD (*i*, Mc, D(Ls)).

SR → S2 Condition: UPD($\ell \neq$ Lr, m > Mc, d $\neq \infty$) Λ buffers empty.
Action: Mc ← m; Lr ← ℓ; Lt ← ℓ; B ← 1;
send FLS(Lt, Mc, FRT);
$\forall$ *i* $\neq$ Lt s.t. C(*i*) = UP then
send UPD (*i*, Mc, D(Ls));
send con(Lr); send ubl.

S1D → S2D Condition: FLS(ℓ = Ld, m, f = DIV).
Action: send adb(Ld, Lt); Lr ← nil;
send FLS(Lt, Mc, DIV);
$\forall$ *i* $\neq$ Lt, Ld s.t. C(*i*) = UP then
send UPD(*i*, Mc, D(Ls)).

S1D → S2R Condition: FLS(ℓ = Ld, m, f = FRT)
Λ buffers non-empty.
Action: send FLS (Lt, Mc, DIV);
$\forall$ *i* $\neq$ Lt, Ld s.t. C(*i*) = UP then
send UPD(*i*, Mc, D(Ls));
Lr ← nil; Ld ← nil.

S1D → S2 Condition: (FLS(ℓ = Ld, m, f = FRT) Λ buffers empty)
$\vee$ FLS(ℓ = Ld, m, f = NFL)
$\vee$ UPD(ℓ = Ld, m = Mc, d $\neq \infty$).
Action: if UPD then f ← NFL;
send FLS(Lt, Mc, f);
if f = FRT then Lr ← Lt;
$\forall$ *i* $\neq$ Lt, Ld s.t. C(*i*) = UP then
send UPD(*i*, Mc, D(Ls));
Ld ← nil send con(Lr); send ubl.

S2D → S3D Condition: FLS(ℓ = Lt, m, f $\neq$ KIL, RST).
Action: $\forall$ *i* s.t. (C(*i*) = UP) Λ U(*i*)' then
send FLS(*i*, Mc, NFL).

S2R → SFR Condition: FLS(ℓ = Lt, m, f $\neq$ KIL, RST).
Action: Lr ← Lt; send con(Lr); send flu;
$\forall$ *i* s.t. (C(*i*) = UP) Λ U(*i*)' then
send FLS(*i*, Mc, NFL).

SFR → S3 Condition: bfl.

Action: send FLS (Lt, Mc, FRT); send ubl; X ← 0.

S3D → SFD Condition: FLS(ℓ = Ld, m, f = FRT).

Action: send fdb.

SFD → SFR Condition: dbf.

Action: Lr ← Lt; send con(Lr); send flu.

Any → SM State Condition: FLS(ℓ = Lr, m, f = KIL).

SD → SM Condition: CHG(ℓ, m, c = FAIL) ∨ lfl(ℓ ≠ Lr).

S1D Action: if FLS then Mc ← m;

S2D ∀ *i* s.t. C(*i*) = READY then C(*i*) ← UP;

S3D ∀ *i* ≠ Lr s.t. C(*i*) = UP then

SFD send FLS(*i*, Mc, KIL);

S2R send rst; send blk;

∀ *i* then (D(*i*) ← ∞; U(*i*) ← 0; F(*i*) ← 0);

B ← 0; R ← 0; P ← 0; X ← 0; start timer T_D.

Any → SM State Condition: (UPD(ℓ = Lr, m, d = ∞) ∨ lfl(ℓ = Lr)) ∧ X

S2D → SM Condition: UPD(ℓ, m > Mc, d = ∞)

∨ UPD(ℓ = Lt, m, D = ∞).

S3D Action: if UPD then Mc ← m;

SFD ∀ *i* s.t. C(*i*) = READY then C(*i*) ← UP;

S2R ∀ *i* ≠ ℓ s.t. C(*i*) = UP then

send UPD(ℓ, Mc, ∞);

send rst; send blk;

∀ *i* then (D(*i*) ← ∞; U(*i*) ← 0; F(*i*) ← 0);

B ← 0; R ← 0; P ← 0; X ← 0; start timer T_D.

S2D → S1 Condition: UPD(ℓ, m > Mc, d ≠ ∞).

S3D Action: Mc ← m; send rst;

SFD ∀ *i* s.t. C(*i*) = READY then C(*i*) ← UP;

S2R Lt ← ℓ; Lr ← ℓ; Ls ← ℓ; Ld ← nil;

B ← 0; R ← 0; P ← 0; X ← 0;

∀ *i* ≠ Lt s.t. C(*i*) = UP then

send UPD(*i*, Mc, D(Ls));

send opr; send ubl; send con(Lr).

SM → S1 Condition: UPD(ℓ, m, d ≠ ∞);
Action: Lr ← ℓ; Lt ← ℓ; Ls ← ℓ; Ld ← nil;
Mc ← m; send opr; send ubl; send con(Lr);
∀ *i* ≠ Lt s.t. C(*i*) = UP then
send UPD(*i*, Mc, D(Ls)).

Sl' → Sl Condition: FLS(ℓ = Lr, m, f = RST).

SD → Sl Condition: Time T_D expired.

SR Action: If FLS then Mc ← m;

SM ∀ *i* ≠ Lr s.t. C(*i*) = UP then
send FLS(*i*, Mc, RST);
send rst; send blk.

Lr, the process performs phase 1, enters S1 and carries on with the normal update cycle.

Receipt of a UPD message with d = ∞ from Lr causes the process to enter the dead state, SD, and halt all sink traffic. If link Lr fails, the process enters the failure root state, SR. Provided that there is not a second failure, the process can leave SD or SR when a UPD message is received with a higher sequence number. If the message is received along Lr, the process moves to S1 and continues as a normal cycle except that the change blocking flag, B, is set in order to suppress any attempt to change paths. If the message is received on a link other than Lr, a node in SD designates this link as its down-tree link, Lt, and the old routing channel as the diversion input channel, Ld. It then transmits the UPD message only along Ld and enters state S1D. If the node is in SB when the UPD message with d = ∞ is received along Lr, the process moves immediately to S1D. Eventually, a UPD message reaches the failure root in state SR. The process designates the receiving link as Lt and performs phase 1 by propagating UPD messages up-tree. If, fortuitously, it does not contain any sink packets the process designates Lt as Lr, immediately performs phase 2 by sending a failure root transfer message (FLS(FRT)) down Lt and enters S2, so continuing as a normal cycle. If, however, the node does contain some sink traffic, it sends a diversion request (FLS(DIV)) down Lt and enters state S2R.

A process in S1D can move in one of three ways. Firstly, if it receives an FLS(FRT) along Ld and the node contains no sink traffic, or

it receives an FLS(NFL) along Ld or a UPD message with $d \neq \infty$ along Ld, it completes phase 1, by transmitting UPD messages along all links other than Ld and Lt. In the case of receiving a failure root transfer message, it transfers the root further down-tree, thus performing phase 2. Otherwise the normal phase 2 is performed by sending an FLS(NFL) down-tree. The process then enters S2. Secondly, if the node receives an FLS(FRT) along Ld while its buffers contain some sink traffic, the node completes phase 1, performs phase 2 by sending a diversion request down-tree and assumes the role of the failure root by entering S2R. Thirdly, if the node receives an FLS(DIV) from Ld, it assigns a diversion buffer to allow traffic to flow from Ld to Lt, completes phase 1, performs phase 2 by sending a diversion request down-tree and enters state S2D. Note, that this assumes that the diversion arrangements can be made instantaneously. If there is any possibility of delay due, for example, to buffer space not being immediately available, then an additional wait state must be inserted between S1D and S2D.

When a process in S2D receives an FLS message from Lt, it performs phase 3. This includes sending an FLS message down Ld. The process then enters state S3D. When the failure root, in state S2R, receives an FLS message from Lt, it also performs phase 3, designates Lt as Lr, starts to flush packets down Lr and enters state SFR until all sink traffic has been flushed. It then sends an FLS(FRT) message down-tree in order to inform the next node, in state S3D, that the diversion is complete, and enters S3. The node receiving this message enters state SFD to await the emptying of its diversion buffer. When this is complete, the process releases the diversion buffer, starts to flush its normal buffers down Lt, which is now designated Lr, and enters SFR. Note that processes, which enter S2 or S3 from a dead state, do so without waiting for FLS messages from all up-tree links. There is no danger in this action because path changes are blocked by the flag B. Indeed, this is a deliberate strategy to enable a diversion to get underway as soon as possible. However, it is necessary, when performing phase 4 in the transition between S3 and SQ, to check not only the UPD received flags, U, but also the FLS flags, F.

Detection of a multiple failure causes the process to drop all packets destined for the sink. It then enters state SM, unless the detection of a multiple failure involved a UPD message with a new sequence number, in which case the process directly enters S1. The

flag X is used to indicate a possibly unflushed up-tree failure. The process will leave state SM when a new UPD message, with a higher sequence number, reaches the node. Finally, if the node remains in states SD, SR or SM for a time greater than TD, it is assumed that no alternative path to the sink exists, and the process resets itself re-entering state SI.

The message handler is not only responsible for the appropriate execution of the FSM but in addition:

(a) records the highest sequence number received, Mn;

(b) maintains the UPD and FLS received flags, U and F;

(c) calculates the distance to sink for each link;

(d) monitors this distance and records the shortest distance link;

(e) maintains the status of each link, automatically bringing the link UP in certain circumstances;

(f) acknowledges UPD messages from cross-tree links with FLS messages if the FLS had not already been transmitted in that cycle.

Most of the time, the routing link, Lr, and the down-tree link, Lt, are the same link. However, they will be different links if the node passes through S1D during the recovery cycle. Lr becomes Lt if and when the process enters SFR. Lt becomes Lr if and when the process enters SQ or S1.

4.5 PROPERTIES OF THE ALGORITHM

The algorithm enjoys a number of properties. These have yet to be rigorously proved. Such proofs would, in the main, be based on the proofs of the original protocol's properties [MERL78].

Property 1: Loop Freedom

(a) At all times, the directed graph, comprising of the nodes and their links defined by the latest con message, forms a set of disjoint trees rooted either at the sink or in a node whose buffers are frozen.

(b) Consider the directed paths formed by the nodes in states S3D and SFD and their links defined by their latest adb message. Then, at

all times, each path terminates at a node which forms part of a tree which is neither rooted at the origin of the path nor at any intermediate node along the path.

The above property guarantees that no loop can be found – even temporarily. It is strongly associated with the properties of the graph formed by the nodes and their routing links, {Lr}, and the graph formed by the nodes and their down-tree links, {Lt}. The latter forms a set of disjoint trees at all times. Unfortunately, the graph based on Lr does not always form a tree although, at the end of an update cycle (when {Lr} = {Lt}), it is guaranteed to form a set of trees. The Lr graph fails to form a tree set when a node is in state S1D and the adjacent node connected to link Lr moves to state S2 from SR. This is because, in so doing, the latter node will denote the same link as Lr thus forming a loop. The loop freedom property is not affected, however, because all sink traffic is frozen when a node is in state S1D, and it cannot leave S1D under these conditions without changing Lr. In any case, this condition lasts only one NTAN message delay before the node in S1D receives an FLS message from Lr, whereupon the node leaves S1D and Lr is changed to equal Lt or nil.

Property 2: Sequentiality

All packets addressed to the sink dispatched from a node, n, either will have arrived at the sink or will have been destroyed before a path change is made by node n.

This property is dependent on Property 1 as well as the correctness of the flushing and multiple failure detection functions.

Property 3: Completion

If a cycle is started with sequence number, m, then, within a finite length of time, this cycle will be properly completed or a link outage will occur in a node having Mc = m. Upon completion of this cycle and until a channel outage or recovery occurs, the set of all nodes, which have some potential path to the sink, form, with their links defined by the latest con message, a single tree rooted at the sink.

Property 4: Recovery

If a lfl or a lrc message is generated within a node, having Mc = m, then an update cycle will be, or has been, started with a sequence number of m+1.

Provided that link failures do not happen too frequently in comparison with the propagation time of the update cycle, Properties 3 and 4 together guarantee that the protocol will always recover and a path will be provided to the sink if it is physically possible.

Property 5: Convergence

Provided that there is no change in the network topology or the link weights, the protocol will tend to minimize the distance of each node to the sink. Within a finite number of cycles, that is not greater than one plus the maximum number of hops across the network, an optimum tree in terms of shortest distance is obtained. Furthermore, the tree is optimized if the sink receives no FLS(FLU) messages down any link for two consecutive cycles having the same sequence number.

The latter part of this property is based on the fact that the FLS(FLU) message indicates a path change up-tree. If there were no path changes then no node could find a better path and, therefore, the tree must be optimized. Because the decision to change routes is made early in the cycle, it may be that a shorter path is not discovered until later in the cycle. The route change would then be made in the next cycle. This is why two cycles clear of flush requests are required before the sink can be sure that the tree is optimized.

With the exception of Property 2, these are very similar to the properties of the non-sequential protocol [MERL79]. The major differences are due to:

(i) the complication of the diversion pathways;

(ii) the different handling of link failures and recoveries;

(iii) the fact that the shortest distance link discovered late in one cycle will not become the routing link until the end of the following cycle;

(iv) the ability of the sink to know when the tree is optimized.

4.6 IMPLEMENTATION

This protocol is ideally suited to MININET. Conversely, the small NTAN message size of MININET is (almost) ideal for this protocol, since it entails many short messages exchanged between adjacent nodes as opposed to the fewer larger messages of other protocols such as the ARPANET routing protocols [MCQU78], [MCQU80].

Each node in the network requires a process for each sink in the network. While the code executed by these processes would be common (with the exception of the sink process for the node itself), separate copies of the variables, defined in Table 4.5, must be stored in RAM for each sink in each node. A ***routing control block (RCB)***, containing the variables described in Table 4.5 plus a pointer to a timer event block for use with timeouts, would be assigned dynamically as the node learns of the existence of the sink node. If the sink does not exist, or the node does not know of its existence, the RCB does not exist. This is the equivalent of state SI in Figure 4.8. Making the reasonable assumption that the absolute maximum number of links connected to a node is 32, the U and F flag array can be stored as two 32-bit set variables. Furthermore, the link status can be stored as two 32-bit set variables representing the links in state UP and state READY respectively. This greatly facilitates fast set operations such as the condition tests for the S1→S2 and S3→SQ transitions in Table 4.7. If Mc, Mh, and each D(ℓ) and W(ℓ) are stored as 16-bit words and the remainder of the variables stored as 8-bit bytes, the contents of Table 4.5, together with a timer event block and a pointer to the timer block, would require 49 + 4.Lmax bytes. For the maximum number of links allowed (i.e. 32), this implies a total memory requirement of 177 bytes per sink. Assuming that the network contains 32 nodes, the total RAM requirement is just over 5.5KB. Using similar assumptions, the sink process would require only 41 bytes of RAM to store the contents of Table 4.2 together with the timer event block and a pointer.

Since Stations must act as a sink in their own right and decide which output channel to use for outgoing packets, they must take part in the routing protocol. However, since a Station cannot switch packets for other nodes, it must transmit a distance of infinity for all destinations other than itself.

The algorithm assumes that the message sequence number may

increase without limit. Of course, in practice, the sequence number range is finite and eventually wraps around and repeats. If the sequence number range is 0 to Mmax, then all comparisons and arithmetic are performed modulo-(Mmax + 1). Consequently, if the difference between two sequence numbers becomes greater than (Mmax + 1)/2, then the polarity of the difference is reversed. This is potentially very hazardous to the routing algorithm and was the principal reason for the introduction of the dead state timer described in Section 4.3.4. The period of this timer, T_D, should theoretically be shorter than the product of minimum time between link failures and the number of increments to the sequence number before wrap-around occurs. However, the period between link failures is very ill defined and has no lower limit. It is assumed that link hold-down is practised by the channel manager (Section 2.3.3). Consequently, because the only likely cause of a close packed sequence of link failures is a single marginally operational link repeatedly failing and recovering, the minimum hold-down period can reasonably be used as the minimum inter-failure period.

One method of increasing the wrap-around distance, without increasing the size of the sequence number field, is based upon the fact that the node will not receive messages with sequence numbers more than 1 or 2 behind its current sequence number, as a new update cycle very quickly supercedes any older cycle. Therefore, each intermediate node's routing process can bias its sequence number comparisons so that the difference between the sequence number of an incoming message and the node's own sequence number has a negative range of only 3 (say) with the remainder of the range considered positive.

The distance carried in a UPD message is, like the sequence number, a variable which has an arbitrary and possibly restrictive maximum value placed on it by the finite dimensions of NTAN messages. One state of the distance variables is reserved to denote infinity. The other states are used to represent the distance range 0 to Dmax. The maximum finite distance, Dmax must be greater than the product of the maximum number of hops across the network and the maximum weight of one hop. This maximum weight should not be smaller than the ratio of the estimated hop delay of the slowest channel to that of the fastest channel.

The formats of the UPD, FLS and CHG messages are shown in Figure 2.8. In terms of required information content, the UPD message

is the most critical as it contains both a sequence number and a distance estimate as well as the sink identity. The sink address field must be, of course, 6 bits long. Therefore, if all three fields were to be squeezed into the 19-bit data field of a NTAN message, there would be only 13 bits left for the distance and sequence fields. This is clearly insufficient. The problem is overcome by utilizing 3 out of the 4 bits of the NTAN class field to make available 16 bits for the distance and sequence fields. The remaining bit of the class field (bit 22) is used to distinguish the UPD message from other S-NTAN messages. Somewhat arbitrarily assigning 10 bits to the distance field gives a distance range of 0 to 1022 with the value 1023 reserved to represent infinity. This leaves 6 bits for the sequence number giving it 64 states. Thus, serial number comparisons would have the numerical range from -3 to +60. The code field of the FLS message carries the possible values, NFL, FLU, DIV, FRT, KIL or RST while the code field of the CHG message carries the values FAIL or REC.

Management modification to the link weights (e.g. manual intervention by an operator) would use MCP (Section 2.6) to transfer the weight update. The sink must be informed of a weight change so that it can start a new update cycle. Furthermore, there may well be more than one link weight to be changed and it would be best to make all the changes prior to starting the update cycle. Therefore, the most convenient mode of operation would be to send a MCP message to the sink listing all the weight updates. The sink then would transmit each new weight to the appropriate node in turn (using MCP). Finally, the sink would start a new update cycle to incorporate the new weights into the distance estimates.

A routing link, identified by the routing management algorithm, can be broken into two components, the channel controller address (internal to the node) and the adjacent node address connected to that channel. The actual packet-by-packet implementation of the routing function is undertaken by special-purpose dedicated processors. This function is distributed between the core processor, which routes the packet to a particular channel, and the channel controller which selects the appropriate adjacent node for the packet (Section 2.3.1). Of course, for a point-to-point channel, the latter function is null and the link is completely specified by the channel address. Routing decisions, within the core of an Exchange, would be implemented by attaching a

particular destination node's queue to a particular channel (Figure 2.18). Disconnecting the queue from any channel has the effect of freezing the buffers. In the Station core, routing is performed by placing the address, of one of its own ports, into the poll list for the output channel appropriate for the destination node of that port's Virtual Connection (Section 5.2.1). Traffic for a particular sink can be frozen by removing all ports connected to that node from the channel poll list. Note, that there are no explicit routing directories used by the high-speed packet handling processors in either the Exchange or the Station.

Chapter 5

STATION ARCHITECTURE

The relatively slow line rates and large packets of the typical wide area network generally result in processing requirements, on each node, that can be adequately met with conventional general-purpose single or multi-processor systems [HEAR70], [ORNS75], [FORN76], [MUEL77]. On the other hand, the required processing rate, within small-packet, high-speed local area networks such as MININET, is too high for a conventional processor structure. One method of improving efficiency is to give several general-purpose microcomputers different jobs within a functionally distributed architecture [FALD76]. However, in order to achieve very high packet handling rates, the individual processors must be designed specifically for their function [MCDE78], [AROZ80]. This design philosophy was adopted in the implementation of a full-speed MININET Station described in this chapter.

5.1 DESIGN REQUIREMENTS

5.1.1 Functional Architecture

The Station may be divided functionally into four parts as shown in Figure 5.1.

(1) The ***port section*** consists of the interfaces to the user devices. Different types of interface can be accommodated. One is the DIM interface described in Chapter 3. Other possible types of port include an IEC-625 [IEC 79] bus interface and a speech port (Section 2.5.2). End-to-end flow control, in the sense of source-sink data rate matching, is handled by the interface protocols (e.g. DIM-CPC described in Section 3.3).

(2) The ***channel section*** consists of up to 8 channel controllers which provide the MININET Channel Service (Section 2.3). For point-to-point channels, this corresponds to the Layer 2 Data Link Service of the OSI Reference Model. However, for multi-node channels, the channel controllers have additional responsibilities

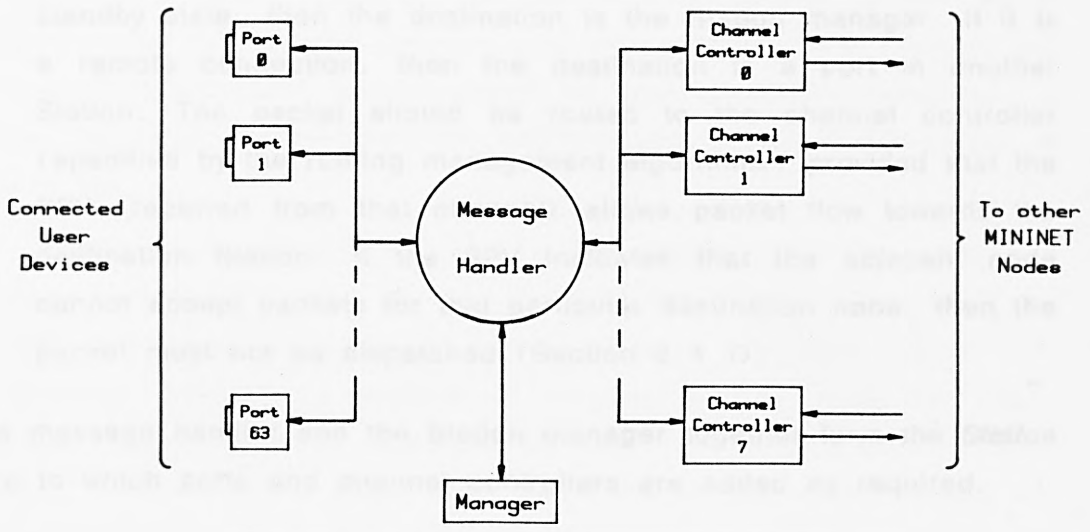


Figure 5.1: Functional Division of the Station

which correspond to Layer 3M. These responsibilities are described in Section 2.3.1. For point-to-point channels this sublayer has no function.

(3) The ***Station manager*** is responsible for all management functions including: Station initialization; establishing and changing Virtual Connections; running the routing algorithm; performance monitoring of the whole Station including ports, channels and the manager itself. The manager can communicate with other node managers by means of MCP (Section 2.6), which provides a message exchange service for the management entities. It also communicates with the operator by means of a console (Section 2.5.1). In addition, the operator can obtain the status of the network, Station, port or channel. It is also possible for user devices to be connected to the Station manager via a port. This enables a computer, for example, to request Virtual Connection changes in the same manner as the operator at the Station console.

(4) The ***message handler*** forms the heart of the Station. Its main job is to route information between the ports, channel controllers and the manager. It must package data entering the network via a port and dispatch the packets towards their destination, as defined by the port's Virtual Connection information described in Section 2.5.1. If it is a local connection, then the destination is another port in the

same Station. If it is a management connection or the port is in the standby state, then the destination is the Station manager. If it is a remote connection, then the destination is a port in another Station. The packet should be routed to the channel controller (specified by the routing management algorithm), provided that the BPV (received from that channel) allows packet flow towards the destination Station. If the BPV indicates that the adjacent node cannot accept packets for that particular destination node, then the packet must not be dispatched (Section 2.4.1).

The message handler and the Station manager together form the ***Station core*** to which ports and channel controllers are added as required.

5.1.2 Speed – Power Characteristics

The maximum design throughput of the channels is in the order of 100k packets per second. This means that at times of peak activity each channel could need feeding once every 10μs and, more importantly, be delivering a packet to the Station once every 10μs. Since there can be up to 8 channels, this implies sub-microsecond packet processing times within the message handler. At the other extreme, it is quite possible that some channels, perhaps using modems, are operating at much lower speeds and have service intervals as long as a few milliseconds. In instrumentation and process control applications, the end-to-end propagation time of a packet is usually more important than throughput. For this reason, greater emphasis is put on minimizing delays than maximizing throughput. In matters such as increase of channel or processing speeds both throughput and delay are improved. However, pipelining of operations and buffering of packets may increase throughput at the expense of increased propagation delay. It is desirable, therefore, in the design of the Station to avoid pipelining and buffering wherever possible.

When moving messages within the Station and when a particular destination (within the Station) is busy, it is important that other traffic is not blocked. For this purpose, each channel controller must be treated as a separate destination because there may be very large differences between channel service intervals. However, since end-to-end flow control is handled by the interface protocols, the ports should not remain busy for any appreciable period and so they can be

lumped together and treated as a single destination. A queue should be incorporated into the port to iron out any fluctuations in the packet delivery rate. The Station manager and the received BPV memory are also destinations which should always be able to accept messages. A queue should be incorporated into the interface towards the manager to ensure this.

The fairness criterion (Section 1.2.3) implies that the ports and channel controllers cannot be connected to the Station core using any method which gives a particular port or channel any fixed priority over the others. Instead, a method of rotating priority must be used. However, because it is desirable to clear the long-distance traffic arriving in the channel controllers, the channels are made equally the highest priority source. Secondly, since the messages generated by the Station manager are important to the proper running of the network, its output should have second highest source priority within the Station. The ports have the third highest priority, which again must be shared equally.

In contrast to the high-speed dedicated processing of the message handler, the Station manager has a large number of relatively sophisticated jobs to perform. However, for the most part, the time constraints on these tasks are less severe. Consequently, the manager is best implemented using a general-purpose microcomputer. Nevertheless, it is advisable to use a fairly powerful 16-bit microprocessor in order to expedite the management operations used to recover from, for example, a channel or port failure.

In order to minimize Station cost and power consumption, it is desirable to use mainly low power Schottky TTL and to avoid multi-layer printed circuit boards wherever possible.

5.1.3 Reliability

In order to meet the requirements of Section 1.2.4, the following precautions have been taken:

(i) Each bus field, be it data or address, uses negative logic and includes an odd parity bit. This particular choice of polarity and parity ensures that failure of a source to turn on forces a bus parity error.

(ii) A soft error should not harden into a permanent error by, for example, causing lockout of a network resource. On the other hand, recurrent failures of resources, such as an intermittent channel, should be hardened to avoid continual re-routing of packets.

(iii) During initialization, the manager should perform a thorough confidence test of the Station.

(iv) During normal Station operation the manager should continually monitor and exercise the operation of the whole Station, and validate the Virtual Connection and routing information stored in the message handler.

5.2 MESSAGE HANDLER DESIGN

5.2.1 Polling Strategy

Earlier versions of the MININET Station [MORL75], [NERI84] connected the ports to the Station core, by means of a bus that used interrupts and a "daisy chain" interrupt acknowledgement system. While the network load was such that this did not cause any real problems in practice, the interrupt approach suffered from several disadvantages. Firstly, the Station core had no control over the input of data into the network. This could lead to temporary blockages if the data could not immediately be delivered to its destination. Secondly, it was impossible to isolate maverick devices connected to the ports. Finally, the acknowledgement chain imposed a fixed priority order on the ports. This contravened the fairness criterion.

In this latest high-speed design each port is polled to find data waiting to be dispatched through the network. The polling operation is performed independently of any transfer to or from the port.

In the initial planning stages of this design, a single poll loop of all connected ports was considered. The data word would then be transferred from the port and, combined with the address fields to form a complete packet, held in a buffer register until the channel through which it was routed was ready to accept a message. The problem, with this pipelined approach, is that other ports are blocked while this packet waits for the channel to become free. This structure was therefore rejected. If the channel availability is gated into the port poll response,

such blockages could be avoided. In order to do this, it would be necessary for the port poller to also poll the channel controllers or maintain a table of channel availability. The problem with such a **source-led** poll is that, in many circumstances, it can be unfair. Assuming that the channel service intervals are independent of the poll loop period (a most doubtful assumption), then a sort of stochastic fairness can be obtained, provided that the ports, connected through the same channel, are placed at equidistant intervals around the loop. If, on the other hand, the ports are bunched together in polling order, then the port at the head of the group has the highest probability of being the first port polled after the channel becomes ready. In any case, the channel service intervals and the poll loop frequency would be fairly constant, leading to beat phenomena giving certain ports precedence. Consequently, this approach was also rejected.

The polling scheme finally adopted is the two-dimensional polling structure shown in Figure 5.2. With the exception of the channel input poller, which remains autonomous, all the polling operations have been combined into a unified ***master arbiter (MA)***. This poller is **destination-led**. Its primary polling loop searches for a destination within the Station that is ready to accept data, while the secondary loops search for a source with a message for that destination. To this end, the connected ports are placed in different poll loops depending on their destination. There is one loop for each channel, plus a loop for the interface to the Station manager, and a loop for the ports connected locally. The intra-Station destination of a port is determined by a combination of Virtual Connection information, which specifies the destination node, and routing information, which determines the output channel for that destination node. When the MA finds a source-destination pair ready and able to communicate, it passes that information on to the ***master transfer controller (MTC)***. This controls the transfer of information along the ***packet bus***, which forms the backbone of the message handler. The packet bus has dual address fields, which allows the MTC to transfer a message from any source to any destination connected to it. The data transferred consists of 3 fields: a message type identifier, a message field (containing either a packet, a NTAN message or an INC message as described in Figure 2.7), and a channel controller address. This last field identifies the destination channel when travelling towards the channels, and the source channel

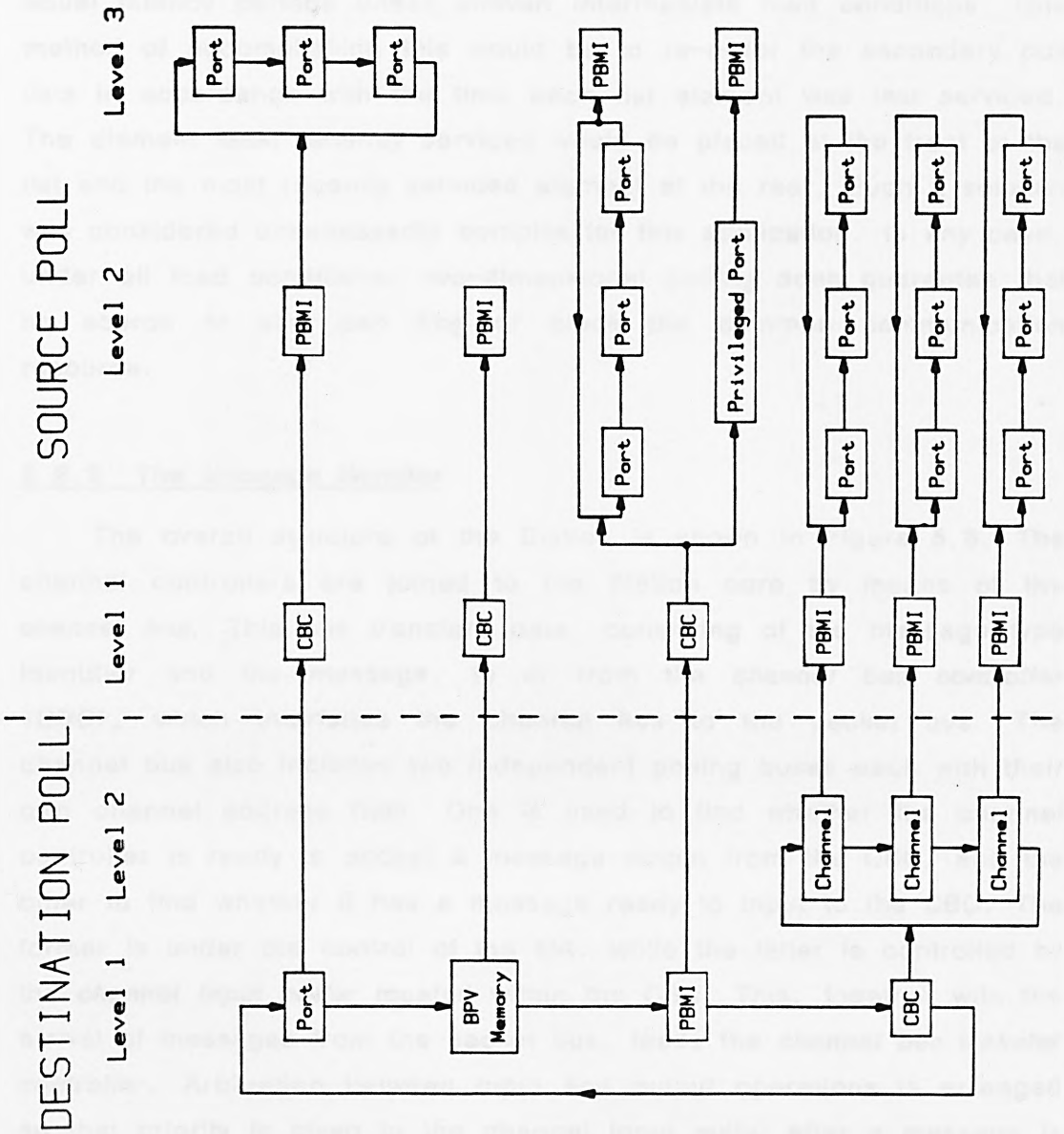


Figure 5.2: Station Polling Sequence

when travelling in the other direction. All these fields include an odd parity bit. These are checked during every transfer along the packet bus. In the rare event of a parity violation, the MTC does not deliver the message but, instead, informs the Station manager for diagnostic and, if possible, recovery purposes.

This design highlights the difficulty, not widely appreciated, of achieving fairness. This two-dimensional polling technique guarantees strict fairness when interconnecting data sources and sinks, via a common resource, under saturation conditions (i.e. all sources wishing to transmit). However, even this polling technique does not guarantee

equal latency periods under uneven intermediate load conditions. One method of accomplishing this would be to re-order the secondary poll lists in accordance with the time each list element was last serviced. The element least recently serviced would be placed at the front of the list and the most recently serviced element at the rear. Such a solution was considered unnecessarily complex for this application. In any case, under all load conditions, two-dimensional polling does guarantee that no source or sink can hog or block the common communication resource.

5.2.2 The Message Handler

The overall structure of the Station is shown in Figure 5.3. The channel controllers are joined to the Station core by means of the ***channel bus***. This bus transfers data, consisting of the message type identifier and the message, to or from the ***channel bus controller (CBC)***, which interfaces the channel bus to the packet bus. The channel bus also includes two independent polling buses each with their own channel address field. One is used to find whether the channel controller is ready to accept a message output from the CBC, and the other to find whether it has a message ready to input to the CBC. The former is under the control of the MA, while the latter is controlled by the ***channel input poller*** located within the CBC. This, together with the arrival of messages from the packet bus, feeds the ***channel bus transfer controller***. Arbitration between input and output operations is arranged so that priority is given to the channel input poller after a message is written to a channel controller, and to the packet bus arrivals after a channel bus read operation. Thus, the resources of the channel bus and CBC are shared fairly between channel transmission and reception. Upon receipt of data transferred from a channel controller, the CBC decodes the message to determine its destination along the packet bus. If it is a user packet and the destination node address field matches the Station's own address, the CBC sends the packet to the port section. If it is a NTAN message containing a BPV update, it is sent to the BPV memory. Anything else, including network packets, INC messages and NTAN routing messages are sent to the Station manager.

The ports are connected via the ***port bus***, which is interfaced to the packet bus by means of the ***port bus transfer controller***. The port bus can be divided into three autonomous sections. The transfer section

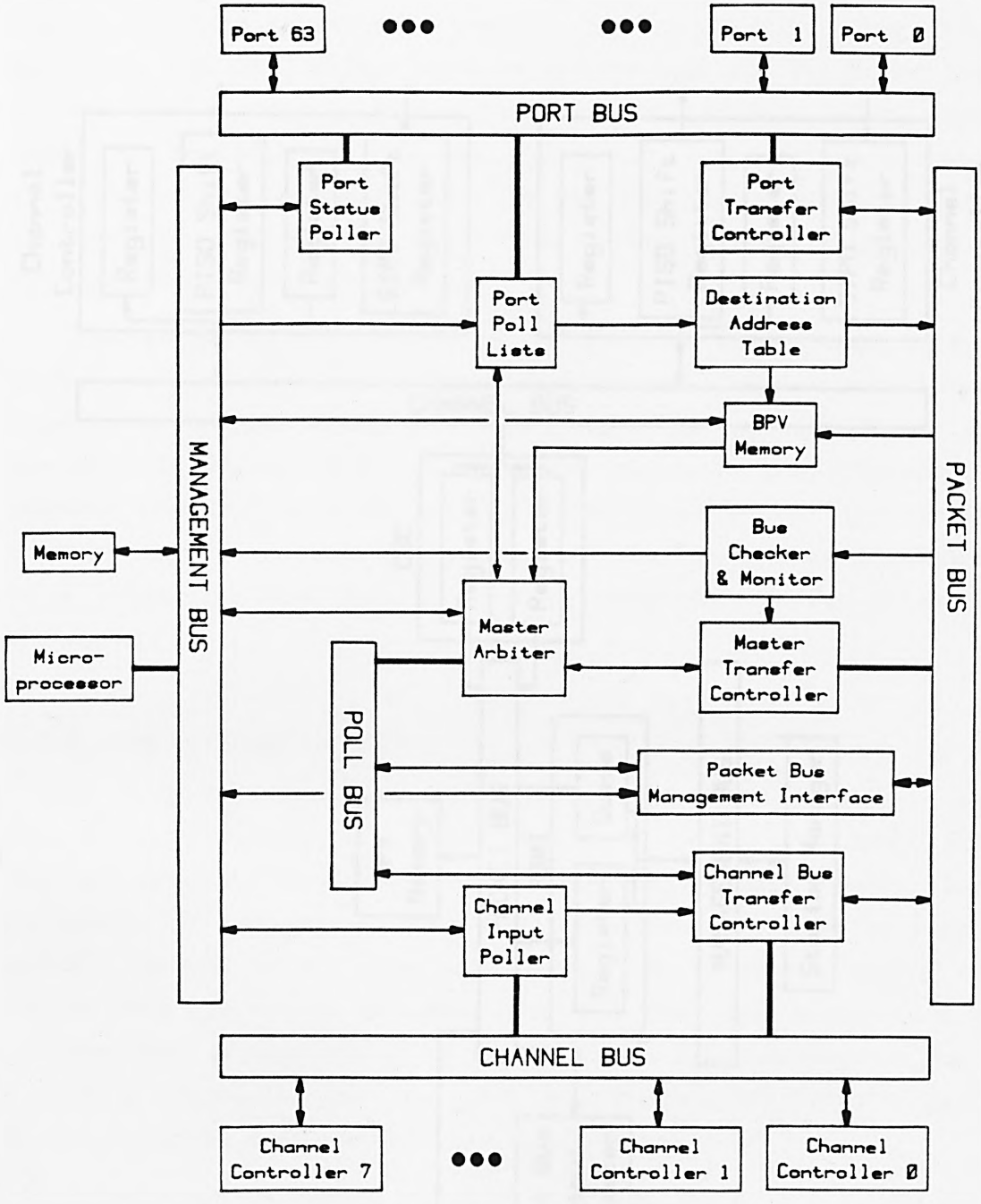


Figure 5.3: Station Structure

moves the data word to or from the ports. The port transfer poller section is used by the MA to search for a port with information to transmit. The port status poller is used to check the health of each port in turn. Errors, such as parity failures or timeouts at the network interface, may be reported by means of this bus.

The locations of FIFO queues and buffer registers within the Station are shown in Figure 5.4. Note that data flows unbuffered, directly from a port to its destination along the packet bus. This is done to minimize

Figure 5.4: Location of Queues and Buffer Registers

transfer delay and simplifies the interface between the MA and the MTC, but means that the port bus must be synchronized with the packet bus for data input operations. Data output along the port bus, on the other hand, can be performed while the MTC is busy with transfers between other elements on the packet bus. The port bus transfer controller uses the destination port address field, within arriving packets, to determine their destination along the port bus.

The Station manager can monitor and control the operation of the message handler by means of the ***management bus***. This is a subset and extension of the microcomputer's system bus. The management bus is connected to the packet bus by means of the ***packet bus management interface (PBMI)***. The PBMI includes a 256-message long input queue to avoid blockages caused by the relatively slow management processor being unable to deal with a burst of messages arriving in rapid succession.

5.2.3 Master Arbiter Operation

In its polling operations, the MA treats the ports individually as sources, but collectively as a single destination. On the other hand, the channel controllers are treated collectively as a single source, but individually as destinations. This has been done because the major unknown, insofar as the poller is concerned, is the service interval of the channels. In normal circumstances, the other destinations have a relatively short service interval of not more than a microsecond or two. A timeout mechanism on the port bus protects the Station from blockages due to deficiencies in user flow control.

In executing the polling sequence described in Figure 5.2, the MA checks each channel controller for its readiness to accept a message. At the same time, it checks whether the PBMI has a message for the channel. If the channel is ready and the PBMI does not have a message, the poller searches, through all the ports connected through that channel, for a port with information to transmit. The poller starts with the port following the last to transmit through that channel. The port poll response is qualified by the content of the BPV memory, which is indexed by the channel and destination Station address. If the MA finds a port or the PBMI ready to transmit (and after waiting, if necessary, for the MTC to become free), it passes the packet bus source and

destination address to the MTC. If the source is the PBMI, the latter supplies the port address, the channel controller address and the address portion of the packet. If the source is a port, this information is supplied by tables within the MA. The MA also records the address of the port following the source port in the poll loop, so that the poll can start with that port on subsequent occasions. The source port will then be the last port to be polled in the loop - effectively becoming the lowest priority source for the channel in question. This rotation of priority guarantees that, if there are **p** ports connected through a particular channel (i.e. **p** ports in its poll loop), then, ignoring any management traffic, each port can obtain at least 1/**p** of the channel throughput. The channel priorities are also rotated so that the Station core resources are shared equally among the channels.

Immediately after successfully finding a channel controller and source ready to transfer information, and passing this information onto the MTC, or after unsuccessfully polling all channels and finding no channel or source ready, the MA returns to destination poll, level 1 (Figure 5.2). It then checks whether the port bus output register is empty. At the same time, it tests whether the CBC or PBMI have data for a port. If both have data and the register is empty then the CBC has priority. If neither have data, the MA polls all locally connected ports, in a similar manner to those remotely connected through the channel controllers. The same procedure is followed for the PBMI as a destination, with the ports connected to the manager being polled. The PBMI, itself, is included as the lowest priority source for test purposes. The BPV memory normally receives updates from the channels but, during initialization, is loaded from the Station manager. Because the BPV memory is, in fact, part of the MA, the MA waits to take part in the transfer operation. This is unlike the procedure with other destinations when it continues to poll ahead while the transfer takes place. Special interlock comparators are used in the MTC and CBC to stop the MA duplicating a transfer.

The MA communicates with the PBMI and CBC by means of the ***poll bus*** which includes packet bus and channel bus destination address fields. The latter field is relayed onwards, as the ***channel output poll address***, along the channel bus. These allow the MA to broadcast the destination currently being polled. The poll bus also includes a number of response lines. One is used for the destination (a channel controller

or the PBMI) to indicate its readiness to accept a message, while two other lines are each used by the PBMI and CBC to indicate that they are holding a message for the specified destination.

5.2.4 Master Arbiter Structure

The structure of the MA is shown in Figure 5.5. For simplicity, the channel controller addresses have been constrained to form a contiguous set starting from zero. This allows the channel poll pointer to be implemented using a simple counter. The channel register contains the highest channel address present in the Station, which is determined by the manager during initialization. This register controls the modulus of the channel counter, which is used to determine the completion of the channel poll loop, and the modulus of the channel poll pointer counter, which determines the currently polled channel controller. The packet bus destination address being polled is derived from the MA controller. Together with the channel poll address, this information is used to index into two memories, the list length memory and the next port list. These provide the length and current position within the corresponding port poll list respectively. The channel buffer register is used to enable a partial overlay of the channel and port poll operations. The next port list supplies the address of the first port in the poll list to the port poll register. This, in turn, indexes into the main port poll list memory. The content of the location, so specified, is the address of the next port in the loop and is fed to the port poll register for the second and subsequent poll cycles. The address of the currently polled port is also fed into the destination address table, which provides its destination port and Station address. The destination Station field, together with the channel controller address, is used to index into the BPV memory, in order to determine whether the adjacent node is accepting any packets for that destination. If this is so and the port has data available, the MA waits for the MTC if it is busy. Then the polled port address is loaded into the source port register and the packet address and output channel address are loaded into the transfer buffer register. The port poll cycle terminates when either the port counter indicates that all ports in the loop have been polled, or a port ready and able to transmit has been found. In the latter case, the next port list is updated with the address of the next port in the loop to achieve the required rotation in port priority.

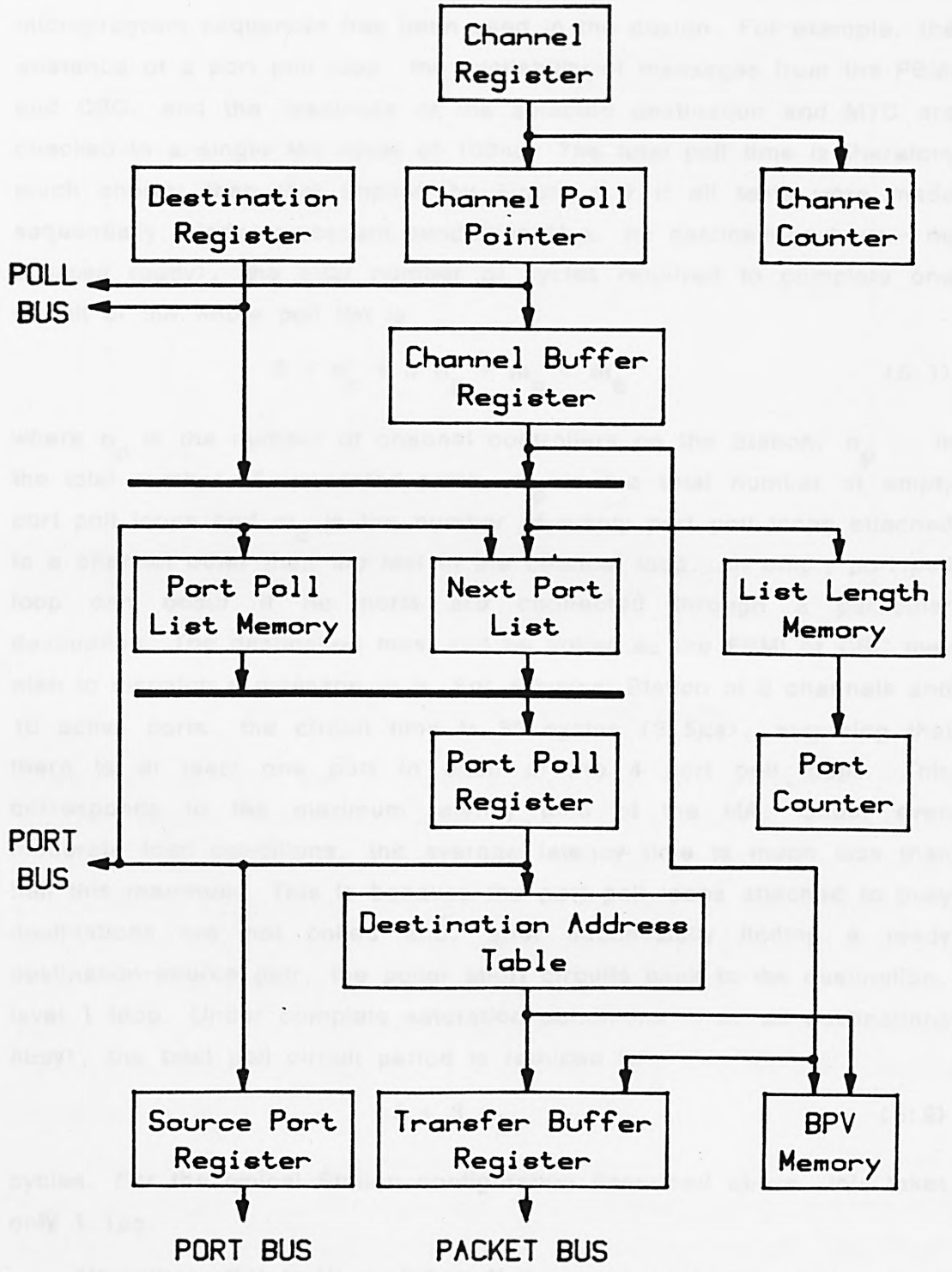


Figure 5.5: Structure of the Master Arbiter

5.2.5 Controller Design

The MA is controlled by a microprogrammed FSM. In order to allow multiple decisions to be taken simultaneously, no conventional microprogram sequencer has been used in the design. For example, the existence of a port poll loop, the availability of messages from the PBMI and CBC, and the readiness of the selected destination and MTC are checked in a single MA cycle of 100ns. The total poll time is therefore much shorter than that implied by Figure 5.2 if all tests were made sequentially. Under quiescent conditions (i.e. no destinations busy - no sources ready), the total number of cycles required to complete one circuit of the whole poll list is

$$3 + n_c + 3\,n_p + m_p + m_c \qquad (5:1)$$

where n_c is the number of channel controllers on the Station, n_p is the total number of connected ports, m_p is the total number of empty port poll loops and m_c is the number of empty port poll loops attached to a channel other than the last in the channel loop. An empty port poll loop can occur if no ports are connected through a particular destination. The destination must still be polled as the PBMI or CBC may wish to dispatch a message to it. For a typical Station of 2 channels and 10 active ports, the circuit time is 35 cycles (3.5μs), assuming that there is at least one port in each of the 4 port poll loops. This corresponds to the maximum latency time of the MA. Under even moderate load conditions, the average latency time is much less than half this maximum. This is because the port poll loops attached to busy destinations are not polled and, after successfully finding a ready destination-source pair, the poller short circuits back to the destination, level 1 loop. Under complete saturation conditions (i.e. all destinations busy), the total poll circuit period is reduced to

$$5 + 3\,n_c \qquad (5:2)$$

cycles. For the typical Station configuration described above, this takes only 1.1μs.

Altogether, this implementation of the message handler uses a total of seven FSMs to control its operation. The design methodology and techniques, developed to implement these and other controllers within the Station, are fully described in [MORL85]. Briefly, the basic approach is as follows. Firstly, the controller's functional requirements are

specified, primarily by means of an ***objective state diagram (OSD)***. In general, this implies that the controller has to be an asynchronous Mealey machine. The controller is then internally partitioned as shown in Figure 5.6. Any required continuous-time functional blocks, such as

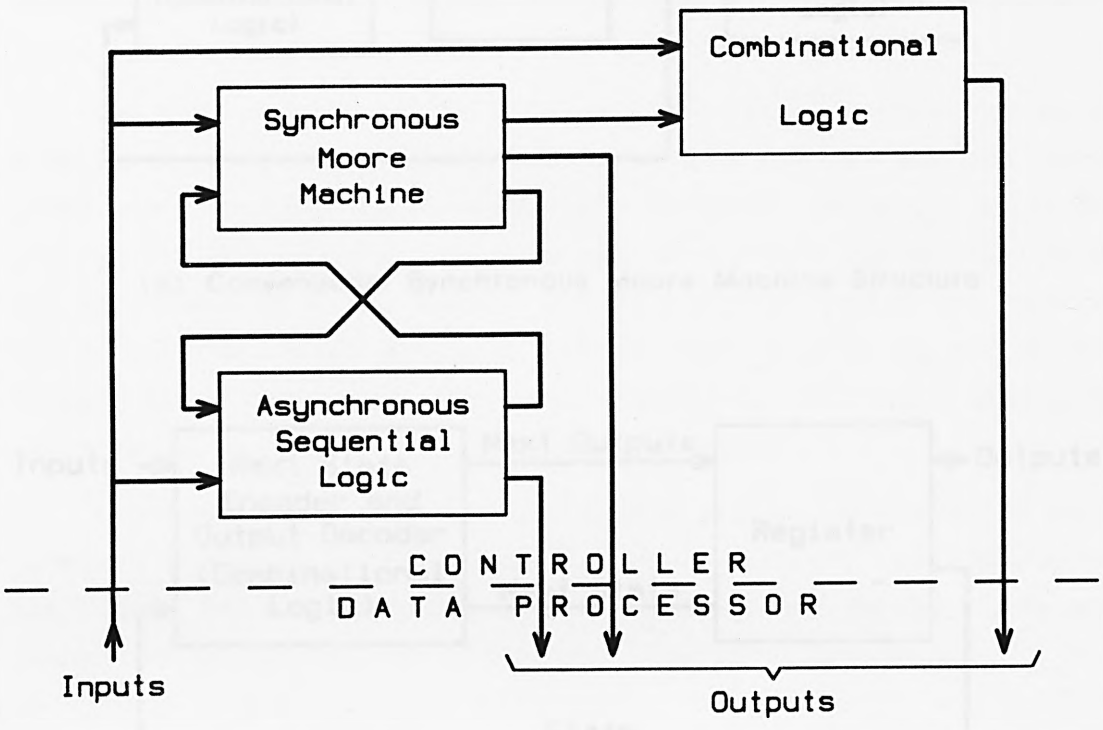
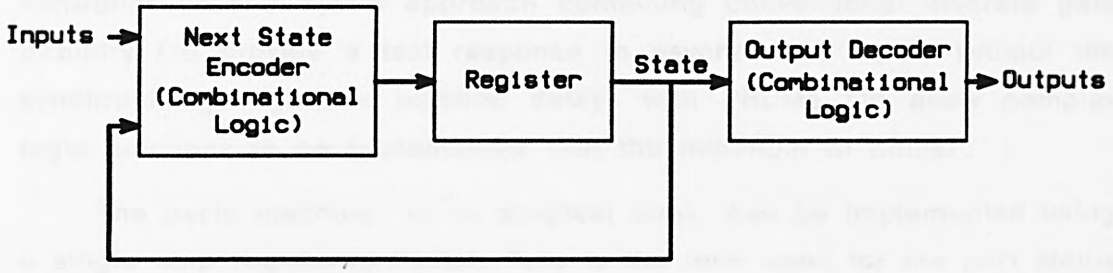


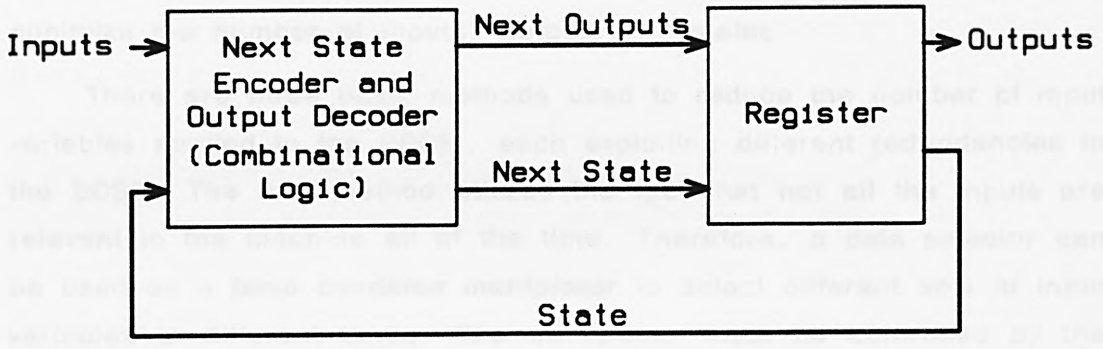
Figure 5.6: General Controller Model

timeout monostables and interlock semaphores, are partitioned into a separate section. Similarly, outputs, which are a function of the inputs as well as the controller state variable, are partitioned into a separate combinational logic block. This leaves, at the heart of the controller, a synchronous Moore machine. The operation of this FSM is specified by means of a ***synchronous controller state diagram (SCSD)***. The rationale, behind this procedure, is that large synchronous Moore machines can be implemented more easily and reliably than large asynchronous Mealey machines.

The synchronous controllers of the channel input poller, MTC and port bus transfer controller are implemented using conventional hard-wired random logic. For the more complex controllers, the Moore machine is implemented using PROMs as arbitrary logic function generators. The conventional output decoder of a Moore machine (Figure 5.7a) is merged with the next state encoder, so that the outputs are computed in parallel with the next state, as shown in Figure 5.7b.



(a) Conventional Synchronous Moore Machine Structure



(b) Moore Machine with Single Combinational Logic Block

Figure 5.7: Synchronous Controller Structure

Both the next state and the machine outputs are passed through a non-transparent register. The former becomes the current state of the machine and is fed back to the input of the next state encoder. Note, that there is no pipeline delay associated with registering the outputs, because they were pre-computed simultaneously with the next state and so appear at the output of the register concurrently with the state.

One disadvantage of using PROMs as combinational logic blocks is that the PROM outputs are undefined, immediately following a change on any of its address input lines, for a period equal to its access time. This means that asynchronous inputs cannot be applied directly to the PROM lest they change value near the clock edge and effectively corrupt the PROM output sampled by the register. The problem can be overcome by passing all inputs, that are not synchronous with the controller clock,

through a synchronizing register. Unfortunately, this can lead to pipeline type delays. For this reason, the design of the channel bus transfer controller uses a hybrid approach combining conventional discrete gate circuitry (to provide a fast response to asychronous inputs without the synchronizing register's pipeline delay) with PROMs (to allow complex logic functions to be implemented with the minimum of chips).

The basic machine, in its simplest form, can be implemented using a single-chip registered PROM. This is the form used for the port status poller and the PBMI queue controller. However, for larger machines such as the MA controller, the finite number of PROM inputs and outputs act as bottlenecks in the design. This is especially true in the case of the inputs where every additional address input doubles the size of the PROM. These restrictions can be ameliorated by techniques aiming to minimize the number of inputs, outputs and states.

There are three basic methods used to reduce the number of input variables applied to the PROM, each exploiting different redundancies in the SCSD. The first method utilizes the fact that not all the inputs are relevent to the machine all of the time. Therefore, a data selector can be used as a ***jump condition multiplexer*** to select different sets of input variables at different times. The multiplexer must be controlled by the FSM and, consequently, the required number of outputs is increased. However, this cost, in terms of PROM size, is small compared with the double-exponentially greater saving on PROM size due to the reduction of inputs.

The second method uses ***pre-encoding*** to concentrate the effective information, stored in sparsely coded input variables, thereby producing a concentrated jump condition code. This utilizes the fact that, in a multi-way branch, the number of branches is far smaller than the number of minterms that can be formed from the number of conditional inputs. For the more complex pre-encoder functions, another PROM can be used as a very powerful pre-encoder. However, the relatively long propagation delays of PROMs usually imply that, to achieve the desired processing speeds, it is necessary to add a pipeline register between the pre-encoder and the main PROM.

The third method is the technique of ***ROM bank swapping***, which utilizes the fact that the states with complex exit conditions are much fewer than the states with uncomplicated branch conditions. This

technique is similar to the normal address extension of a memory selecting between two or more memory banks, except that different banks have a different mix of state variable feedback and inputs connected to their address pins. For example, one 512-word PROM bank may have 7 address inputs connected to the state variable, leaving 2 inputs for conditional inputs, while another 512-word bank may have only 2 address inputs connected to the state variable, with the other 7 free to be connected to conditional inputs. Thus, the first bank can be used for up to 128 different states but the exit conditions of these states cannot be very complex. On the other hand, the second bank can distinguish between only 4 states but the large number of inputs allows it to handle complex multi-way branches.

All these techniques can be used in conjunction to produce very agile controllers relatively economically. An example of the power of these methods is the master controller for the half-duplex channel controller, which uses one 1K x 4-bit PROM as a pre-encoder, one 8 to 1 data selector as a jump condition multiplexer, and two 512 x 8-bit PROMs with one 1K x 8-bit PROM, arranged in two banks, as the main encoder. Altogether, this controller has a total of 19 inputs and 207 states. In two of these states, the controller makes an 18-way branch as a function of 15 inputs in a single 100ns clock cycle. In the case of the MA controller, three 1K x 16-bit ROM banks are used, together with a 2-pole 4-way jump condition multiplexer, to form a FSM operating at 10MHz. Its SCSD contains 45 states controlled by a total of 18 inputs. A large number of states have relatively complex exit conditions, involving up to a 7-way branch as a function of 8 inputs.

Two methods are used to reduce the number of PROM outputs. The first exploits the fact that many outputs are ***mutually exclusive*** (i.e. never asserted at the same time). Consequently, a binary decoder can be used to decode an encoded output from the registered PROM. For example, a 3 to 8 line decoder can be used to provide up to 7 mutually exclusive outputs. Note, that one output code has always to be reserved for the states where none of the outputs are asserted. The other method of output variable reduction is that of ***output variable sharing***, where the values of the state variable for each state are chosen so that some of its bits can be used directly as outputs. This is especially useful with partially defined outputs (i.e. outputs whose value is only specified, in the SCSD, for some of the states). The select controls for the jump

condition multiplexer are an example of this type of output, because the select output is only defined in a state whose exit conditions are a function of the multiplexer output.

Minimization of the number of states of the machine is doubly beneficial, as it reduces both the number of PROM address inputs and the number of PROM outputs. Long strings of states to provide time delay can be avoided by the use of monostables or, if greater accuracy is required, by a counter. The latter can also be used as a loop counter, which allows the removal of states generating repetitions of the same sequence of events. The counter can be loaded, from the controller, with different starting values to vary the number of loops or the delay length. Quite dramatic reductions in the number of states can be achieved with these techniques. In contrast, the technique of ***state merging*** is useful to save just a few states. It is frequently used to reduce the total number of states to a power of two, so that the number of state variables is reduced by one. This technique takes advantage of the fact that the PROM outputs, that are destined to become the controller outputs after passing through a register, can be made a direct function of the inputs without the inputs necessarily affecting the next state variable. This may be thought of as a ***pseudo Mealey machine*** because, from the point of view of the SCSD, this is similar to the specification of a Mealey machine. However, since the output passes through a register, it is strictly still a Moore machine.

5.3 THE STATION MANAGER

5.3.1 MINTOS - The Management Operating System

The management of the Station is undertaken by a number of concurrent tasks running inside a 16-bit microcomputer. Inter-task communication is accomplished by means of an operating system especially developed for MININET. In the first operating system developed, the conventional semaphore primitives "Signal" and "Wait" were utilized to handle task synchronization and queue management [LAUE75]. While this operating system was quite powerful, experience with the snail network suggested that a more powerful and flexible means of inter-task communication was required. In particular, tasks needed to be able to wait for a whole number of different types of events at the same time. Consequently, a new operating system, named MINTOS,

was developed for the new Station design. Tasks communicate by means of FIFO ***event queues (EQs)***. The elements of the queues are messages of arbitrary size and structure. These might be quite short, such as a network packet or a timeout notification, or relatively large, such as an MCP message. A task can wait for messages to be placed in one or more EQs. It does this by calling a ***multi-wait*** primitive specifying a list of EQs that it wishes to test. The message, at the front of the first non-empty EQ on the list, is passed to the task. If all the specified EQs are empty, then the task is suspended and attached to each empty EQ. If more than one task is waiting on the same EQ, then a queue of tasks is formed. Note, that by virtue of the multi-wait primitive, a task can be in a number of queues at the same time. Of course when, eventually, a task is given a message it is removed from all task queues.

This structure enables event servicing to be ordered chronologically by placing event messages in the same EQ, and/or ordered in a prioritized fashion by placing the messages in different EQs. Because more than one task can wait on the same EQ, buffer pools are very easily implemented as EQs which are initialized with a full complement of usable message buffers by MINTOS.

Timer management is obtained using a primitive which relays a message to a specified EQ after a specified time has elapsed. Thus, event timeouts are implemented simply by sending a delayed timeout message to the same EQ as will be used by the expected event notification message, and then waiting on this EQ. If the expected message does not arrive within the timeout interval, the timeout message will be received by the task instead. On the other hand, if the expected message arrives first, the timeout message must be deactivated by using a special primitive which removes the timeout message from the timer or event queue.

Interrupt handlers are implemented as tasks. Each level of interrupt is enabled by a task executing a ***wait-interrupt*** primitive which causes the task to wait until the interrupt is active. It is the responsibility of the task to service and clear the interrupt condition before re-enabling the interrupt. The interrupt handlers communicate with other tasks using the standard EQ primitives.

5.3.2 Management Tasks

The overall view of the management system including tasks, EQs and the major data bases is shown in Figure 5.8. For simplicity, buffer

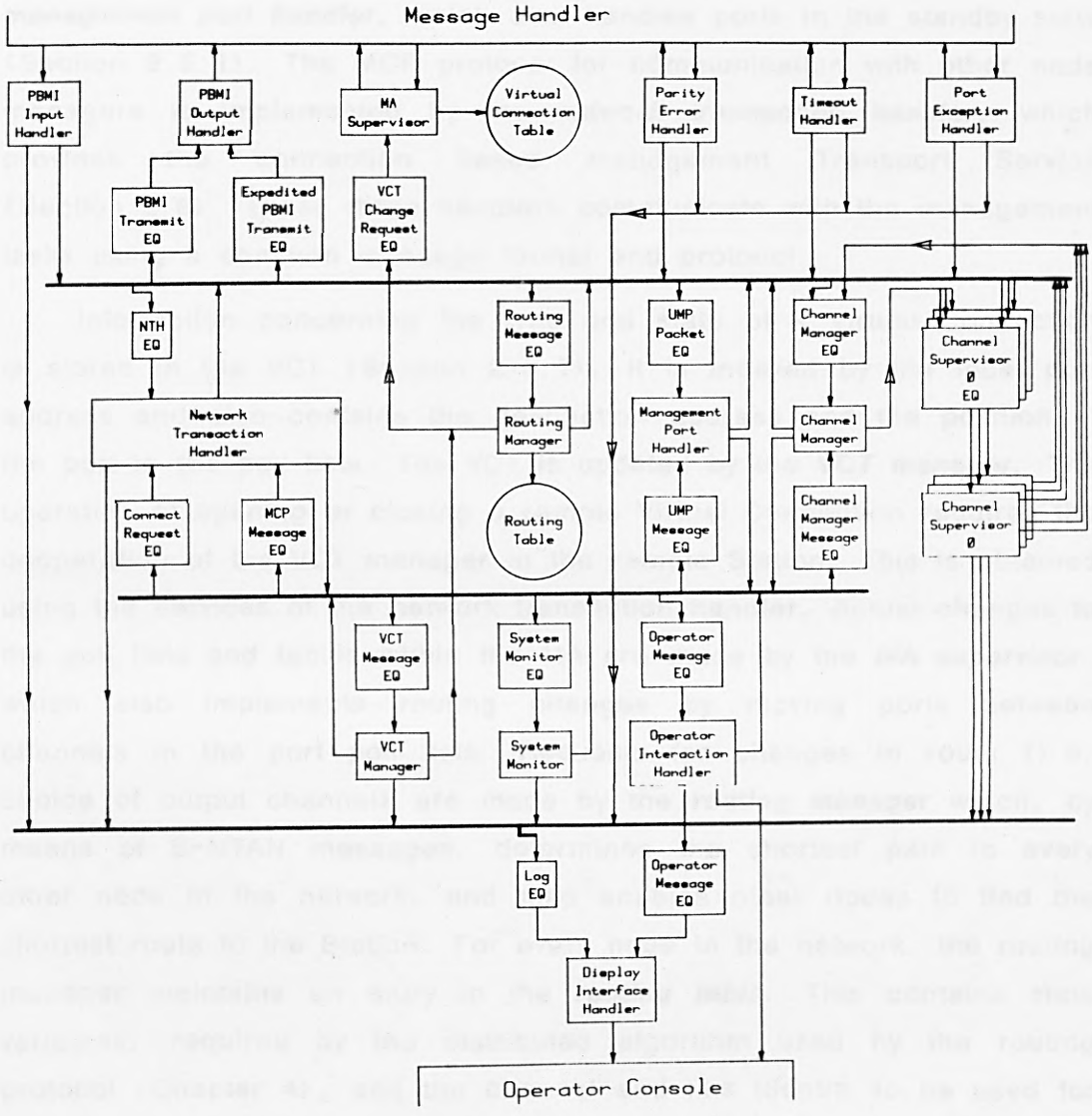


Figure 5.8: Management Task and Event Queue Interconnection

pool EQs are not shown. Incoming messages from other network nodes, management ports and channel controllers are decoded by the ***PBMI input handler*** and sent to the appropriate EQ. The ***PBMI output handler*** performs a similar operation in the other direction. It has two EQs for outgoing messages. The lower priority EQ is used for normal transmission, while the other is used for expedited transmission of urgent messages such as those concerned with rerouting and channel recovery operations. Dialogue with the user via a terminal or dedicated

front panel is handled by the ***operator interaction handler***. Messages to the operator are formatted by the device-dependent ***display interface handler***, while the device-dependent portion of the input processing is performed by a procedure contained within the operator interaction handler. Communication with user computers is managed by the ***management port handler***, which also handles ports in the standby state (Section 2.5.1). The MCP protocol for communication with other node managers is implemented by the ***network transaction handler***, which provides the connection based management Transport Service (Section 2.6). These three handlers communicate with the management tasks using a common message format and protocol.

Information concerning the type and state of a Virtual Connection is stored in the VCT (Section 2.5.1). It is indexed by the local port address and also contains the destination address and the position of the port in the poll lists. The VCT is updated by the ***VCT manager***. The operation of opening or closing a remote Virtual Connection requires the cooperation of the VCT manager in the remote Station. This is obtained using the services of the network transaction handler. Actual changes to the poll lists and tables within the MA are made by the ***MA supervisor***, which also implements routing changes by moving ports between channels in the port poll lists. Requests for changes in route (i.e. choice of output channel) are made by the ***routing manager*** which, by means of S-NTAN messages, determines the shortest path to every other node in the network, and also enables other nodes to find the shortest route to the Station. For every node in the network, the routing manager maintains an entry in the ***routing table***. This contains state variables, required by the distributed algorithm used by the routing protocol (Chapter 4), and the channel and link identity to be used for the first hop. This latter information is used by the MA supervisor when opening new Virtual Connections and by the network transaction handler when transmitting network packets.

During Station initialization, the existence and type of each channel controller present in the Station is established. For each channel controller, a ***channel supervisor*** task is dynamically created. The channel supervisors are specific for each type of channel controller. They supervise channel operations such as synchronization and extract statistical information on errors and packet flow. The ***channel manager*** has overall responsibility for the operability of all the channels and

communication with adjacent nodes. It also practises the "hold-down reflex" to suppress intermittent channels, described in Section 2.3.3.

Upon initialization and prior to activating any task, MINTOS checks the RAM used by the management tasks. The first task activated performs read/write tests on the poll lists, destination address table and BPV memory within the MA. Only after this task has completed its confidence tests of the operation of the message handler, does normal operation of the Station commence. Exception conditions occurring in the hardware message handler are handled by the ***parity*** and ***timeout error handlers.*** Exception conditions occurring at the port-user interface are handled by a separate task, the ***port exception handler.*** After attempting to clear the fault condition and get packets flowing again, the exception handlers and channel supervisors send a report to the display interface handler and update the ***system log table.*** The latter enables operating statistics to be accumulated and extracted by the operator. Background monitoring of the Station's memory is performed by the ***system monitor*** which continually validates the checksum of the VCT and routing table, and compares the content of the MA lists with that contained in the VCT.

5.4 STATION IMPLEMENTATION

This Station design was constructed using catalogue parts. Most logic was implemented using low-power Schottky TTL components, while the time critical portions used standard Schottky TTL. The Station core occupies six 220mm x 233mm Eurocard two-layer printed-circuit boards which are shown in Figure 5.9. The top three boards together contain the MA, MTC, port bus controllers and the packet bus parity checkers. The lower left board is the PBMI, the lower centre board the CBC, while the lower right board contains the management computer. The size could be reduced considerably by incorporating custom integrated circuits into the message handler.

In addition to the DIM and speech ports, a half-duplex channel controller was designed and constructed (Figure 5.10). The physical layer is implemented on the upper left half circuit board using a MS43 ternary transmission code [FRAN68] operating at 20M Baud. The data link controller occupies the rest of the boards. This achieves throughputs of 140k packets per second over a separation distance of

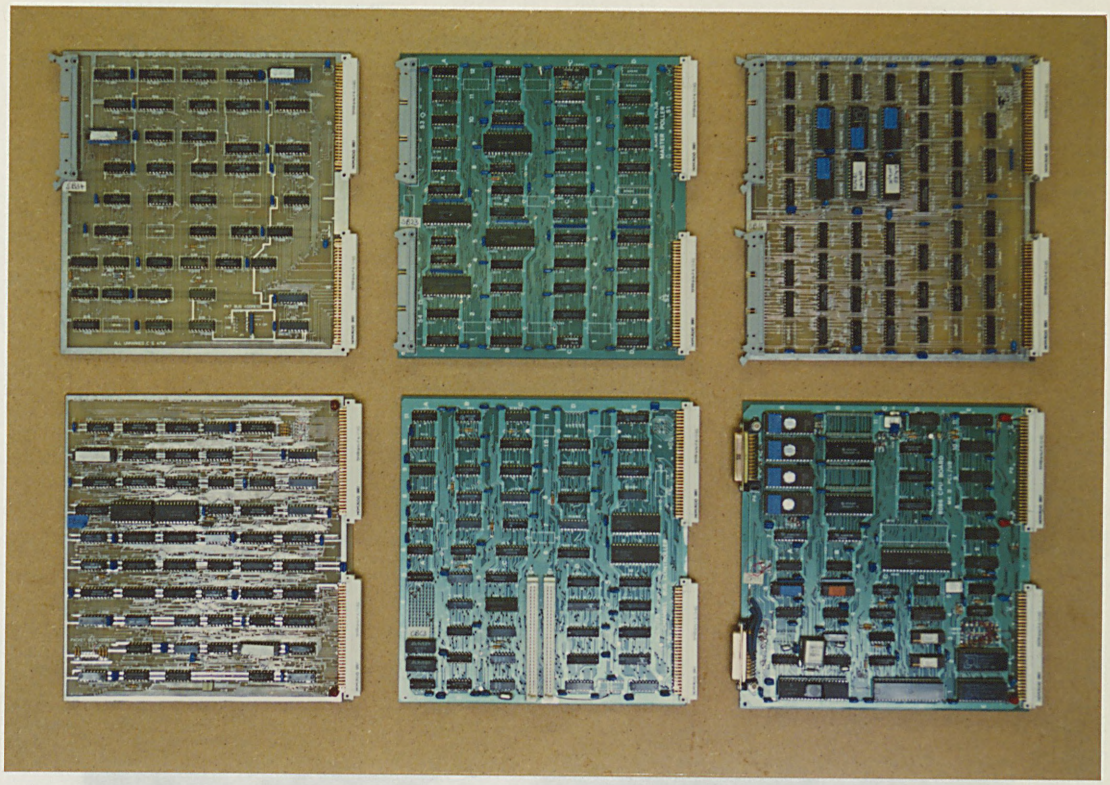


Figure 5.9: The Station Core Circuit Boards

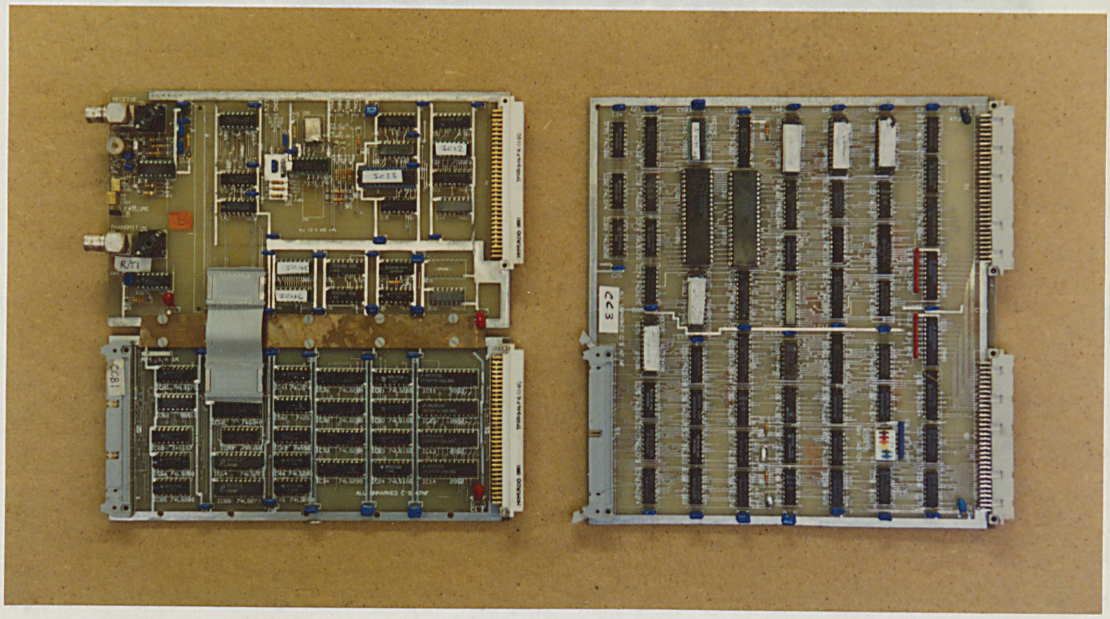


Figure 5.10: Half-Duplex Channel Controller

100m. The design allows other Physical Layer implementations to be attached to the same data link controller. A complete Station, equipped with 5 DIM ports and one channel controller is shown in Figure 5.11. The two coaxial cables are the physical medium of the channel, while

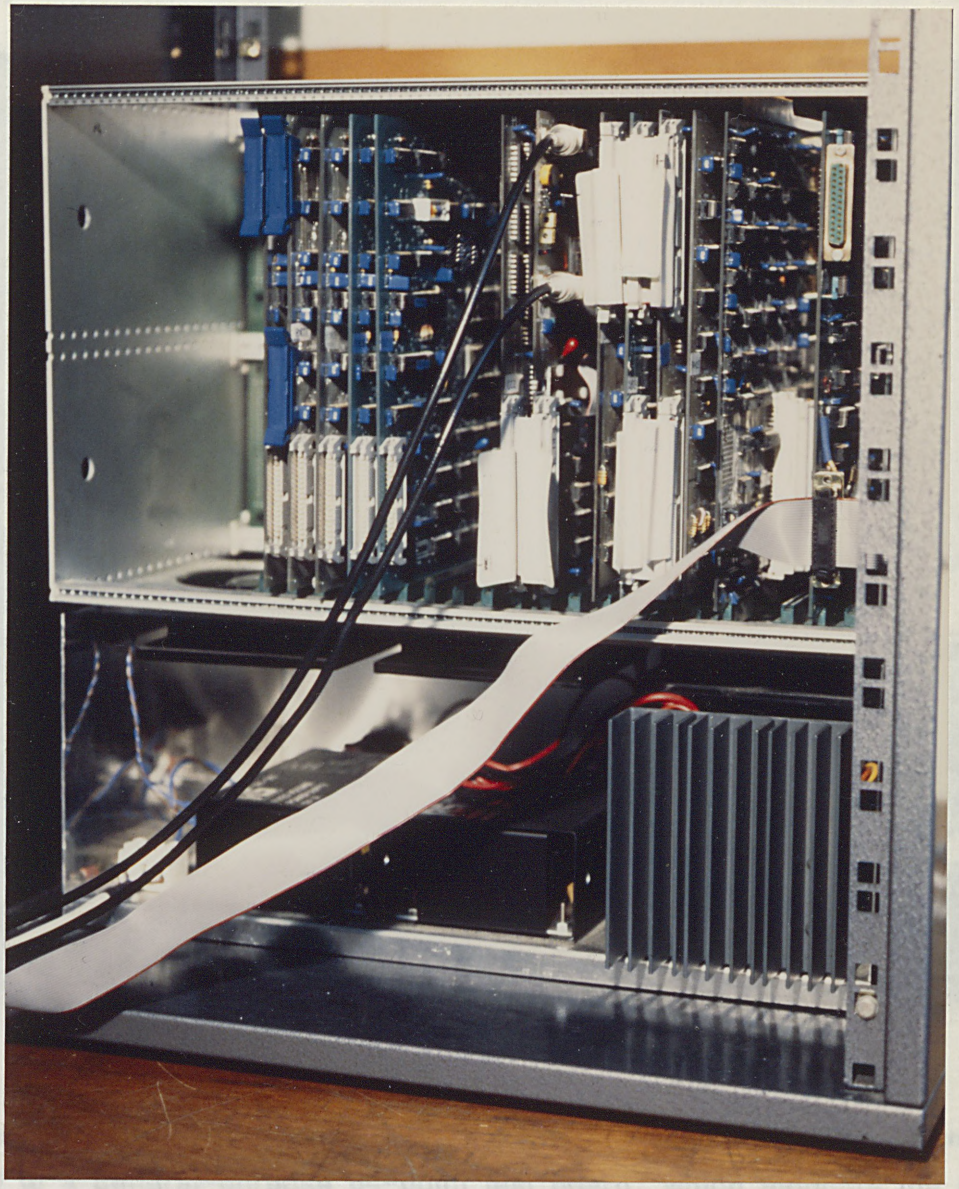


Figure 5.11: Rear View of a MININET Station

the flat cable connects the operator's console to the Station manager. With locally connected ports, user throughputs have been measured corresponding to over 700k packets per second in the message handler.

The management processor is an Intel 8086 microprocessor. Its system bus conforms electrically to the Multibus IEEE-796 standard [IEEE83]. However, in common with the rest of the Station mechanics, the more robust Eurocard mechanical standard was used for this bus. The operating system kernel was implemented in assembler, while the tasks are, for the most part, written in Pascal.

Testing of the complete Station, or one or more of its sub-systems, during development was greatly facilitated by a system

monitor and exerciser (Figure 5.12) based upon an Apple

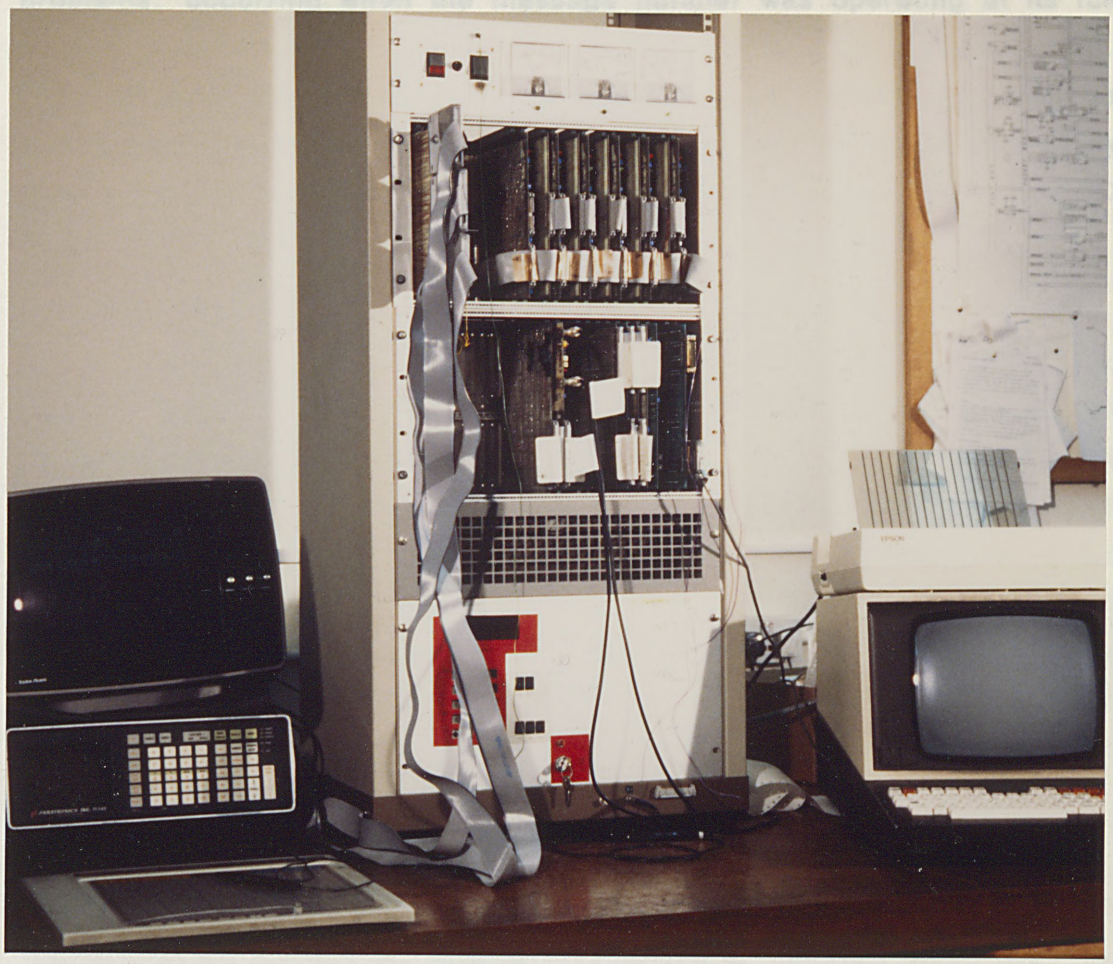


Figure 5.12: Station Development Monitor and Exerciser

microcomputer. Each of the five buses along the Station backplane is wired to a general-purpose 64-point test board, which monitors the logic level on each bus line and, in addition, optionally drives the line to a high or low level. These cards, which can be seen at the top of the rack in Figure 5.12, are interfaced to the microcomputer via an IEC-625 bus. A graphical display and tablet, also interfaced to the microcomputer, provide the designer with a continual monitor of the activity on each bus of the Station and a "soft keyboard" input. The latter enables the designer to emulate the controller or a slave on each bus during sub-system development, or merely to observe the activity along each bus during system integration. With the controlling program written in a mixture of assembler and interpreted Pascal, the microcomputer samples and displays the state of all buses at a rate of approximately 5 updates per second. The system clock of the message handler is controlled by the microcomputer, so that the Station can be "walked" at

a rate slow enough for the human observer to follow the action. Timing problems, occurring when the message handler was operating at its full speed of 10MHz, were investigated using further monitoring circuits, which enabled a multi-channel logic analyser to be connected to any bus at the press of a single "soft button". Testing was further facilitated by the built-in ability to monitor the internal state of the MA, PBMI and CBC controllers via the management bus.

Chapter 6

CONCLUSIONS

6.1 PROJECT STATUS

The development of MININET originated as an in-house project to solve actual laboratory communication problems. From this small beginning, it grew to become a research project in its own right. The problems encountered and solved during the development of the network encompassed a wide variety of topics, ranging from clock recovery at 20M Baud to operating system design, from interface noise rejection to sequential routing algorithms, from high-speed controller design to the hierarchical modelling implications of multi-node channels, and so on. Despite all the successful and innovatory design work undertaken, the project is currently moribund.

It is instructive to examine the reasons for this situation. Certainly, the intermittent nature and level of funding was not commensurate with the size of the project. In particular, the funding hiatuses made it almost impossible to keep a project team intact. However, this was not the whole reason for its failure. Originally, the network was envisaged as a low-cost solution to the needs of remote instrumentation. In particular, the Station was expected to be relatively simple and, therefore, an inexpensive unit in comparison with the Exchange. Since it was expected that there would be two to three times as many Stations as Exchanges in a network, this would have kept the total cost of a network installation within reasonable bounds. Of course, the magnitude of many of the design problems, such as routing, congestion control and maintaining fairness had not been forseen. As evidenced in Chapter 5, the ramifications of the fairness criterion, together with the required packet processing speed, made the Station design much more complicated than the structure envisaged at the outset of the project [MORL75]. Almost certainly, it would have been better to have concentrated the design effort on the Exchange, with its store-and-forward network relay function, instead of on the Station. The level of complexity in the Exchange design would not have been that much greater than in the final

Station design. Furthermore, the amount and complexity of the managerial software development task had been underestimated.

In summary, the network design became too complex, and consequently too expensive, for its application areas.

This situation could, possibly, have been avoided by lowering the quality of the MININET Service. The transparency requirement could have been relaxed and some network awareness been expected from the network users. Then a larger packet could have been used to achieve the same information throughput with a lower packet rate. This, in turn, would have reduced the processing requirements of the nodes allowing simpler packet processing hardware. However, the ultra-transparent service requirement has always been fundamental to the design of the network and makes its Network Service distinctive in comparison with other networks. Another approach would have been to reduce the speed requirement allowing the use of less specialized hardware, although this would have narrowed the network's area of application. This would have excluded, for example, audio signal processing. Another requirement that could have been relaxed was the fairness criterion. It appears innocuous enough, a requirement that no just-minded network designer could refuse. However, the complexity of the Station design, as already discussed, is evidence of its cost. Had it been ignored, a much simpler Station could have been designed. It is doubtful if the user would have noticed much difference under normal network load conditions - especially if most of the traffic was operating in handshake mode. However, if the network was heavily loaded with a lot of burst mode traffic, the level of unfairness could have been very noticeable.

Another approach to reducing the cost of the network would have been to simplify the network's internal structure, while retaining all the essential elements of the MININET Service. If all the links had had the same capacity, then the routing protocol could have been somewhat simplified, the flow control could have been made more efficient and the Station's polling structure greatly simplified (Section 5.2.1). An even more extreme simplification would have been not to allow an arbitrary network topology. If the network was restricted to being a single bus or ring, then none of the Network Layer store-and-forward functions would have been required. In particular, there would have been no need for a routing protocol, congestion control or Exchanges and the Station design would have been very straightforward. The access method used by the

Cambridge Ring [HOPP77], with its small fixed-length PDUs, anti-hogging mechanism and short access time, is especially suited to real-time instrumentation applications (Section 1.3.1). A MININET type service could very easily be built on top of a Cambridge Ring protocol allowing fully transparent communication between user devices without any knowledge of the network protocols.

The advances, both in semiconductor technology and in design methodology [MORL85A] have made it, not only possible, but almost the norm to integrate large parts of digital systems into custom integrated circuits. This has made feasible the routine use of algorithms, which hitherto were considered economically impractical. Examples range from the use of error correction coding in domestic compact disc players to the sophisticated embellishments of the IEEE 802.5 token passing ring standard [IEEE85]. Had the design of the MININET nodes been targetted towards a custom silicon implementation, a smaller and more reliable product would have been the result. Much of the high-speed special-purpose circuitry would have mapped very efficiently onto a custom-chip implementation. However, the very wide interconnecting buses would have caused problems as far as pin-out is concerned, and narrower multiplexed buses would have been preferable even though they may well have reduced somewhat the packet processing throughput.

In conclusion, it seems that the most economical means of providing the MININET Service would be to adopt a fixed ring topology using a Cambridge Ring type access method and implement the design using custom silicon techniques.

6.2 FURTHER RESEARCH TOPICS

Notwithstanding the previous discussion concerning the economic viability of the full-blown MININET conception, the development of the network revealed several areas of interest which would profit from further research. One is that of congestion control. This seems to be something of a "Cinderella" subject in comparison with routing and other network protocols, with only a few networks having an effective deadlock avoidance scheme. The solution proposed for MININET, using active backpressure flow vectors (Section 2.4.1), has problems of efficiency if the channels, connected to a node, have very different throughputs. The rate of generation of BPVs, which could be triggered by traffic

arriving and departing through the high-speed channels, would tend to saturate the low-speed channels. An approach based upon the uses of different maximum BPV rates for channels of different capacity could solve the problem, but it is not clear how this could be implemented efficiently. A second problem arises if the buffer partitioning scheme used by MININET is applied to another larger network, where the large number of nodes makes the partitioning scheme, based on the destination node address, impractical. A more general buffer class scheme could be used [MERL80] but, unlike many of the existing buffer class based algorithms, some sort of fairness should be maintained.

In any case, it would be useful to investigate the performance of active flow vectors (in comparison with the passive flow vectors used by TYMNET [RIND79]) together with the various buffer handling algorithms proposed in Section 2.4.1. The efficiency of the algorithm, in terms of buffer utilization, packet delay and packet throughput could be examined as a function of the length of the reservoir, hysteresis and overflow zones.

Another very interesting area of future research is that of high-speed packet switch design. The basic algorithm, for the output switch of the Exchange, consists of a two-dimensional poll. The primary poll loop checks the readiness of each output channel, while the secondary loops check the availability of packets in each destination queue attached to that channel (Figure 2.18). The latter must be qualified by the BPV received from the appropriate channel. The starting point of each secondary poll must be rotated, following the dispatch of a packet along the channel, in a fashion analogous to that of the port poll loops within the Station (Section 5.2.1).

The packet processing rate of the Exchange would be limited primarily by the speed of this poll, if it is implemented directly. Alternatively, the secondary poll loop could be replaced by a vector approach. Let D be the set of all destination queues that are non-empty, R_i be the set of all destinations reached via channel i, as determined by the routing algorithm, and V_i be the latest received flow vector from channel i. Then the set of queues attached to output channel i with packets ready and able to be transmitted, T_i, is given by

$$T_i = D \cap R_i \cap V_i \qquad (6:1)$$

If T_i is considered to be ordered in a circular fashion, then it can be considered as a circular list of available queues. Searching this list, from some given starting point, until a non-zero entry is found, is equivalent to performing the secondary poll loop function described above, provided that the starting point is set to the element following the last queue to have a packet transmitted.

Since each set can be physically represented by means of a 64-bit binary array, the operation corresponding to (6:1) can be implemented very simply by an array of 3-input AND gates to form the T_i array. The second operation, that of rotating the array to the correct starting point, can be performed by means of a combinational circular shifter. The final operation, that of selecting the first queue with a packet available for transmission, can be implemented by means of a priority encoder. To the address output from the priority encoder, the starting address must be added, in order to identify the queue selected for next output. These processes are illustrated in Figure 6.1. Note that all the operations involve only combinational logic. Therefore, the complete queue poll loop could be performed in one cycle for each channel, independently of the number of queues attached to that channel. With a maximum of 64 nodes, a direct implementation of this algorithm is certainly feasible within a custom chip design. The rotator array would be implemented in 6 stages, each consisting of a 64-pole 2-way data selector. There is, however, a fairly horrendous connection mapping between each stage, which would increase the interconnection area on the chip, and increase the propagation delay through the rotator. The priority encoder would almost certainly be implemented as a two stage tree taking 8 bits at a time, or even a 3 stage tree taking 4 bits at a time. The total propagation delay of this combinational logic block would be quite considerable. Even so, with the speed of modern integrated circuit technologies, it is unlikely to exceed the delay in eliciting the channel controller's readiness for transmission. In any case, pipelining could be used within the block in order to improve its throughput.

Another implementation, which could be used to reduce the amount and complexity of the combinational logic, is to multiplex the vectors to the priority encoder, say, 16 bits at a time. The rotation of the array is then performed, initially to the nearest 16 bits, by means of the order in which the segments are fetched from memory and finally by a 32-bit input, 16-bit output barrel shifter. Because the shifter has to be initially

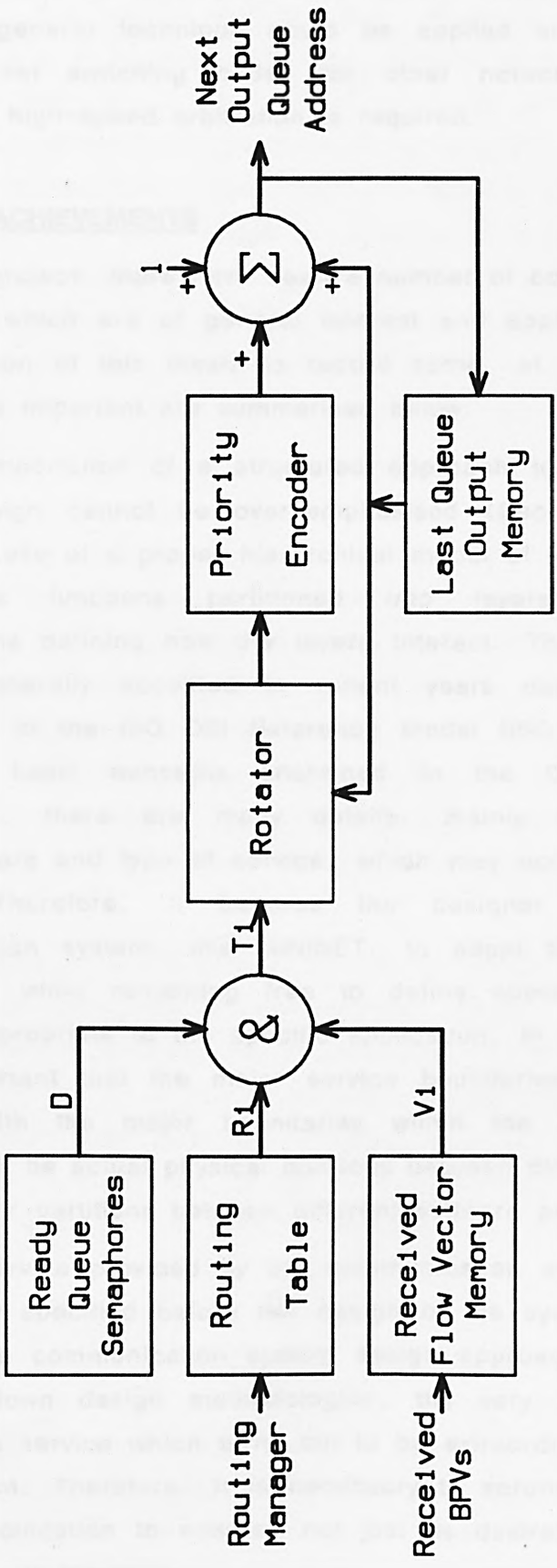


Figure 6.1: Vector Implementation of Exchange Output Poll

loaded. It would take up to 5 cycles to perform a complete poll for one channel. This general technique could be applied to the design of high-speed packet switching nodes for other networks, or indeed anywhere where high-speed arbitration is required.

6.3 PROJECT ACHIEVEMENTS

From this project, there have been a number of concepts, lessons and techniques which are of general interest and applicability. It has been the intention of this thesis to record some, at least, of these items. The more important are summarised below:

(1) The importance of a structured approach to communication system design cannot be over-emphasised (Section 2.1). This entails the use of a proper hierarchical model of the system, with its various functions partitioned into layers and service specifications defining how the layers interact. These ideas have become generally accepted in recent years due to the wide acceptance of the ISO OSI Reference Model [ISO 84]. However, while the basic concepts enshrined in the OSI model are indisputable, there are many details, mainly concerning the specific layers and type of service, which may not suit a specific network. Therefore, it behoves the designer of a closed communication system, like MININET, to adopt the OSI layered philosophy, while remaining free to define sublayers and their services appropriate to the specific application. In particular, it is vitally important that the major service boundaries in the model coincide with the major boundaries within the implementation, whether they be actual physical divisions between different pieces of hardware, or partitions between different software processes.

(2) The service provided by the communication system needs to be carefully specified before the design of the system can start. This modern communication system design approach shares, with other top-down design methodologies, the very real danger of specifying a service which turns out to be extraordinarily expensive to implement. Therefore, it is necessary to scrutinize carefully a service specification to ensure, not just its desirability, but most importantly, its feasibility.

(3) Transparency, taken to the point of invisibility, is probably the

most notable aspect of the MININET Service. It embodies the assumption that user devices are primarily concerned with communicating with each other on a point-to-point basis and, therefore, they do not wish to be bothered with details of the network connection. Of course, in situations, such as office automation applications, where the users wish to communicate with a large number of other users in the network, it is entirely proper that they are more aware and concerned with the presence of the network. However, in an instrumentation environment, this is usually not the case and the MININET concept of network transparency could well be applied to other networks designed for that type of application.

(4) The network layer problems, raised by the inclusion of multi-node channels within MININET, are very similar to those encountered in internetworking [SIND83]. The solutions adopted by MININET are of direct relevance to the design of a global network. Note, that the functionality of the additional 3M sublayer introduced to the MININET model (Section 2.3.1) is more than a mere subnetwork service equalization layer. Wherever possible, global network protocol functions are devolved to the sublayer in order to ease the processing burden of the main global network layer entities.

(5) A suitable choice of packet size in the design of a network is influenced by two major considerations. Firstly, there are the consequences of the service requirements of the network. The ramifications, on the quality of service, of segmenting or blocking must be taken into account. In the case of MININET, these requirements (Section 1.2.2) forced a very small packet size on the network design. The second consideration is that of channel and buffer memory utilization efficiency. Section 2.2.2 discusses techniques for estimating the suitability of a particular packet size given an estimate of the distribution of user message sizes. Note, that the choice of a variable packet size generally improves efficiency, as far as channel utilization is concerned, but has little effect on buffer efficiency since most allocation implementations reserve buffer space up to the maximum packet length.

(6) Data-link protocols are increasingly implemented in dedicated hardware processors. Indeed this is mandatory, where very high

frame processing rates are required. It is desirable, therefore, to adopt a simple data link protocol. MLP, described in Section 2.3.2, is ideal for such applications, provided that the frames are of fixed length. Despite the simplicity of its implementation, it is, nevertheless, very robust, thanks to its property of treating corrupted frames in an identical manner to error messages. Furthermore, its single sequence number field makes it efficient as far as header overhead is concerned.

(7) The anti-congestion flow-control algorithm, described in Section 2.4.1, guarantees fairness and freedom from store-and-forward deadlock, without resorting to dropping packets. The destination node based buffer partitions restrict the use of this algorithm, in its present form, to smaller networks. However, it could be easily extended if hierachical addressing and routing is used.

(8) The management transport protocol, MCP, described in Section 2.6, goes to great lengths to maintain reliability, even if the quality of the Packet Delivery Service is poor due to some malfunction of the network hardware. In particular, the problem of unacknowledged packets, during the closure of a conversation, has been overcome.

(9) DIM is a relatively straightforward interface (Chapter 3) which, nevertheless, has a very useful capacity. It is, therefore, very suitable for interfacing data conversion equipment to a computer or linking computers together, even over extended distances of 10m or more, where data rates in order of mega-bits per second are required.

(10) At the beginning of the project, it was not at all clear whether it would be possible to devise a routing protocol which maintains intrinsic packet sequency. The routing algorithm described in Chapter 4 achieves this aim, with packets being dropped only in exceptional circumstances. This distributed algorithm constructs a set of guaranteed loop-free trees, based on each destination in the network. A quad-phasic update cycle is required to flush old packet pathways, so maintaining packet sequency. This is obtained using only small messages exchanged between adjacent nodes. Many of the connection-orientated wide area networks, such as TYMNET

[RIND76], TRANSPAC [DANE76] and SNA [AHUJ79], use an approach where routes are fixed at connection establishment time and all connections, passing through a link or node which fails, are automatically closed. These networks could well be improved by the use of the routing algorithm developed here.

(11) It has been shown that fairness in network protocol design is much more difficult to achieve than at first appears. Indeed, there are a number of ways in which fairness can be defined leading to the concept of different degrees of fairness. The fairness criterion was restricted to be essentially an anti-hogging requirement. Destination-led, two-dimensional polling structures (Section 5.2.1) were used to achieve fairness in the Station design. In Section 6.2 a high-speed implementation of a similar algorithm, for the Exchanges, was proposed.

(12) In order to achieve the required performance of the Station message handler, it was necessary to develop agile microprogrammed controllers. The PROM based design technique which evolved (Section 5.2.5) can be used in a wide variety of applications.

(13) The management processes needed a method of inter-task communication, where a task could wait for a number of events simultaneously. The resulting operating system (Section 5.3.1) provides this by means of a multi-wait primitive, where each task can wait on a number of FIFO event queues which are ranked in priority order by the task. This relatively simple concept provided an operating system which has remarkable versatility in real time applications.

Overall, the project has made contributions to many different branches of communication engineering and system design.

REFERENCES

[ABRA70] Abramson, N., "The ALOHA system", AFIPS Conf. Proc., vol. 37, pp. 281-285, 1970.

[ACAM83] Acampora, A.S., M.G. Hluchyj and C.D. Tsao, "A centralized-bus architecture for local area networks", Proc. Int. Commun. Conf. (ICC), Boston, Mass. (IEEE), June 1983.

[AHUJ79] Ahuja, V., "Routing and flow control in Systems Network Architecture", *IBM System J.*, vol. 18, no. 2, pp. 298-314, 1979.

[AJMO83] Ajmone Marsan, M. and M. Gerla, "Tokenet - a token based local area network", Proc. Mediterranean Electrotechnical Conf. (MELECON '83), vol. 1, paper A1.01, Athens, Greece, May 1983.

[AROZ80] Arozullah, M., S.C. Crist and J.F. Burnell, "A microprocessor based high-speed space-borne packet switch", *IEEE Trans. on Commun.*, vol. COM-28, no. 1, pp. 7-21, January 1980.

[BSI 79] British Standards Institution, *A Digital Input/Output Interface for Data Collection Systems*, BS4421, London, England, 1969.

[BOEH64] Boehm, S.P. and P. Baran, "On distributed communications - II: digital simulation of hot-potato routing in a broadband distributed communications network", RAND Corp. Rep. RM 3101-PR, August 1964.

[BOEH69] Boehm, B.W. and R.L. Mobley, "Adaptive routing techniques for distributed communication systems", *IEEE Trans. on Commun. Technol.*, vol. COM-17, pp. 340-349, June 1969.

[BRAN72] Brandt, G.J. and G.J. Chretien, "Methods to control and operate a message-switching network", Proc. Symp. on Computer Communs., Networks and Teletraffic, Polytechnic Institute of Brooklyn, p. 263, April 1972.

[CAIN74] Cain, G.D., R.C.S. Morling and P.M. Stevens, "Comparisons of fixed and variable packet size for data communication", PCL Technical Memorandum MN-2, Polytechnic of Central London, November 1974.

[CAIN78] Cain G.D. and R.C.S. Morling, "MININET: A local area network for real-time instrumentation and control", Proc. 3rd Conf. on Local Computer Networking, Minneapolis, Minnesota, October 1978.

[CHLA80] Chlamtac I. and W.R. Franta, "Message-based priority access to local networks", *Computer Communs.*, vol. 3, no. 2, pp. 77-84, April 1980.

[CHON82] Chong-Wei Tseng and Bor-Mei Chen, "D-net. A new scheme for high data rate optical local area network", Proc. GLOBECOM '82, Miami, Florida, November 1982.

[COHE81] Cohen, D., "On holy wars and a plea for peace", *Computer*, vol. 14, no. 10, pp. 48-54, October 1981.

[DANE76] Danet, A. et al., "The French Public Packet Switching Service: the TRANSPAC network", Proc. 3rd Int. Conf. on Computer Commun. (ICCC), pp. 251-260, Toronto, Canada, August 1976.

[DAVI73] Davies, D.W. and D.L.A. Barber, *Communication Networks for Computers*, Appendix, Wiley, 1973.

[DAVI79] Davies, D.W. et al., *Computer Networks and their Protocols*, Wiley, 1979.

[DIJK59] Dijkstra, E.W., "A note on two problems in connexion with graphs", *Numerische Mathematik*, vol. 1, pp. 269-271, 1959.

[ECMA82] European Computer Manufacturers Association, *Network Layer Principles - Final Draft*, ECMA/TC24/82/18, Geneva, Switzerland, January 1982.

[ESON72] ESONE Committee, *CAMAC - a Modular Instrumentation System for Data Handling - Revised Description and Specification*, EURATOM Std. EUR4100e, Commission of the European Communities, Luxembourg, 1972.

[ESON76] ESONE Committee, *Serial Highway Interface System (CAMAC)*, EURATOM Std. EUR6100e, Commission of the European Communities, Luxembourg, 1976.

[FALD76] Faldella, E., G. Neri and T. Salmon, "A two-microprocessor implementation of a MININET Station", Proc. 2nd. Symp. on Micro Architecture (Euromicro'76), pp. 305-310, Venice, Italy, October 1976.

[FALD78] Faldella, E., G. Neri and T. Salmon-Cinotti, "High-speed data link control in MININET", Trends and Applications 78: Distributed Processing, NBS Report, Gaithersburg, Maryland, May 1978.

[FARM69] Farmer, W.D. and E.E. Newhall, "An experimental distributed switching system to handle bursty computer traffic", Proc. ACM Symp. on Problems on the Optimization of Data Commun. Systems, pp. 1-33, Pine Mountain, Georgia, October 1969.

[FLET73] Fletcher, J.G., "The Octopus computer network", *Datamation* pp. 58-63, April 1973.

[FORD62] Ford, L.R. and D.R. Fulkerson, *Flows in Networks*, Princeton Univ. Press, 1962.

[FORN76] Forney, G.D., and J.E. Vander Mey, "The Codex 6000 series of intelligent network processors", *Comp. Comm. Review*, vol. 6, no. 2, pp. 7-11, April 1976.

[FRAN68] Franaszek, P.A., "Sequence state coding for digital transmission", *Bell System Tech. J.*, vol. 47, no. 1, pp. 143-157, January, 1968.

[FRAS74] Fraser, A.G., "Spider - an experimental data communication system", Proc. Int. Commun. Conf. (ICC), (IEEE) pp. 21F.1-21F.10, 1974.

[FRAS75] Fraser A.G., "A virtual channel network", *Datamation*, pp. 51-56, February 1975.

[FRAS79] Fraser, A.G., "DATAKIT - A modular network for synchronous and asynchronous traffic", Proc. Int. Commun. Conf. (ICC), (IEEE), pp. 20.2.1-20.2.3, June 1979.

[FUCH70] Fuchs, E. and P.E. Jackson, "Estimates of distributions of random variables for certain computer communications traffic models", *Commun. of the ACM*, vol. 13, no. 12, pp. 752-757, December 1970.

[FULT72] Fultz, G.L., "Adaptive routing techniques for message switching computer communication networks", Univ. of California, Los Angeles, Rep. UCLA-ENG-7352, July 1972.

[GERL80] Gerla, M. and L. Kleinrock, "Flow control: a comparitive survey", *IEEE Trans. on Commun.*, vol. COM-28, no. 4, pp. 553-574, April 1980.

[GRAN78] Grangé, J.L. and M.I. Irland, "Thirty nine steps to a computer network", Proc. 4th Int. Conf. on Computer Commun. (ICCC), pp. 763-769, Kyoto, Japan, September 1978.

[GRAN79] Grangé, J.L., "Traffic control in a packet switching network", IRIA Report SCH 618, May 1979.

[HAFN74] Hafner, E.R., Z. Nenadal and M. Tschanz, "A digital loop communication system", *IEEE Trans. on Commun.*, vol. COM-22, no. 6, pp. 877-881, June 1974.

[HAIN82] Hainich, R., "An improved Ethernet for real-time applications", Proc. Real Time Data '82, 2nd Int. Symp., pp. 257-265, Versailles, France, 1982.

[HEAR70] Heart, F.E. et al., "The interface message processor for the ARPA computer network", AFIPS Conf. Proc., vol. 36, pp. 551-567, June 1970.

[HEGE78] Heger, D., Communication methods in line sharing systems and a comparison of their performance", *IITB-Mitteilungen*, pp. 41-48, 1978.

[HOPP77] Hopper, A., "Data ring at Computer Laboratory, University of Cambridge", Local Area Networking Workshop, NBS Report No. 500-31, pp. 11-16, Gaithersburg, Maryland, August 1977.

[HOPP79] Hopper, A., "A maintenance of ring communication systems", *IEEE Trans. on Commun.*, vol. COM-27, no. 4, pp. 760-761, April 1979.

[IEC 79] International Electrotechnical Organization. *An Interface System for Programmable Measuring Instruments*, IEC Standard 625. Geneva, Switzerland, 1975.

[IEC 81] International Electrotechnical Organization. *Process Data Highway for Distributed Process Control Systems, Part 1: General Description and Functional Requirements*, February 1981.

[IEEE83] Institute of Electrical and Electronics Engineers. *IEEE Standard: Microcomputer System Bus*, IEEE Std. 796, 1983.

[IEEE84] Institute of Electrical and Electronics Engineers. *IEEE Standards for Local Area Networks: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, IEEE Std. 802.3, 1984.

[IEEE85] Institute of Electrical and Electronics Engineers. *IEEE Standards for Local Area Networks: Token-Passing Ring Access Method and Physical Layer Specifications*, IEEE Std. 802.5, 1985.

[IEEE85A] Institute of Electrical and Electronics Engineers. *IEEE Standards for Local Area Networks: Token-Passing Bus Access Method and Physical Layer Specifications*, IEEE Std. 802.4, 1985.

[ISO 79] International Standards Organization. *High Level Data Link Control Procedures - Elements of Procedures*, ISO 4335, Geneva, Switzerland, 1979.

[ISO 84] International Standards Organization. *Information Processing Systems - Open Systems Interconnection - Basic Reference Model*, ISO 7498, Geneva, Switzerland, 1984.

[ISO 87] International Standards Organization. *Basic Reference Model for Open Systems Interconnection: Connectionless-Mode Transmission*, ISO 7498/Add 1, Geneva, Switzerland, 1987.

[KAHN72] Kahn, R.E. and W.R. Crowther, "Flow control in a resource-shared computer network", *IEEE Trans. on Commun.*, vol. COM-20, no. 3, pp. 539-546, June 1972.

[KAWA83] Kawashima, M., R. Yatsuboshi and Y. Mochida, "High capacity LAN by LSI and fiber optics", 2nd Int. Workshop on VLSI in Communs., (Unpublished), Saratoga Springs, New York, June 1983.

[KLEI74] Kleinrock, L. and W.E. Naylor, "On measured behavior of the ARPA Network", Network Information Center, Network Measurement Note no. 18, July 1974.

[KROP72] Kropfl, W.J., "An experimental data block switching system", *Bell System Tech. J.*, vol. 51, no. 6, pp. 1147-1165, July 1972.

[LAUE75] Lauesen, S., "A large semaphore based operating system", *Commun. ACM*, vol. 18, no. 7, pp. 377-389, July 1975.

[LIMB82] Limb, J.O. and C. Flores, "Description of Fasnet - a unidirectional local area communication network", *Bell System Tech. J.*, vol. 61, no. 7, pp. 1413-1440, September 1982.

[MCDE78], McDermid, J.A., "COMFLEX - a high speed packet switch for inter-computer communication", Proc. European Computing Congress (Eurocomp78), pp. 187-204, London, U.K., May 1978.

[MCQU77] McQuillan, J.M. and D.C. Walden, "The ARPA Network design decisions", *Computer Networks*, vol. 1, pp. 243-289, 1977.

[MCQU77A] McQuillan, J.M., "Routing algorithms for computer networks - a survey", Nat. Telecomm. Conf., session 28, paper 1, pp. 28:1.1-28:1.6, December 1977.

[MCQU78] McQuillan, J.M., G. Falk and I. Richer, "A review of the development and performance of the ARPANET routing algorithm", *IEEE Trans. on Commun.*, vol. COM-26, no. 12, pp. 1802-1811, December 1978.

[MCQU79] McQuillan, J.M., "Interactions between routing and congestion control in computer networks", Int. Symp. on Flow Control in Computer Networks, session 2, paper 3, Paris, France, February 1979.

[MCQU80] McQuillan, J.M., I. Richer and E.C. Rosen, "The new routing algorithm for the ARPANET", *IEEE Trans. on Commun.*, vol. COM-28, no. 5, pp. 711-719, May 1980.

[MERL78] Merlin, P.M. and A. Segall, "A failsafe distributed routing protocol", EE Pub. No. 313, Dept. Elec. Eng., Technion, Haifa, May 1978.

[MERL79] Merlin, P.M. and A. Segall, "A failsafe distributed routing protocol", *IEEE Trans. on Commun.*, vol. COM-27, no. 9, pp. 1280-1287, September 1979.

[MERL80] Merlin, P.M. and P.J. Schweitzer, "Deadlock avoidance in store-and-forward networks - I: store-and-forward deadlock", *IEEE Trans. on Commun.*, vol. COM-28, no. 3, pp. 345-354, March 1980.

[METC76] Metcalfe, R.M. and D.R. Boggs, "ETHERNET: distributed packet switching for local computer networks", *Commun. of the ACM*, vol. 19, no. 7, pp. 395-404, July 1976.

[MORL74] Morling, R.C.S., "MININET: a real-time packet switching network", presented at Minicomputers in Data Communs., (PCL) Firenze, Italy, May 1974.

[MORL75] Morling, R.C.S. and G.D. Cain, "MININET: a packet-switching minicomputer network for real-time instrumentation", Proc. A.I.M. Int. Meeting on Minicomputers and Data Communs., Liege, Belgium, January 1975.

[MORL78] Morling, R.C.S. et al., "The MININET internode control protocol", Proc. Symp. on Computer Network Protocols, pp. B4.1-B4.6, Liege, Belgium, February 1978.

[MORL83] Morling, R.C.S., "DIM: a network compatible intermediate interface standard", *Interfaces in Computing*, vol. 1, no. 2, pp. 117-144, 1983.

[MORL85] Morling, R.C.S., "System controller design", *Electronic Product Design*, Vol. 6, Part 1: no. 7, pp.55-58, July 1985; Part 2: no. 8, pp. 45-48, August 1985; Part 3: no. 9, pp. 81-83, September 1985.

[MORL85A] Morling, R.C.S. and A.J. Carter, "Hierarchical system design and the engineering workstation revolution", presented at the Int. Computer Graphics User Conf., London, U.K., February 1985.

[MUEL77] Mueller, D.J., "Microcomputers decentralize processing in data communication networks", *Computer Design*, pp. 81-88, October 1977.

[NERI77] Neri, G., R.C.S. Morling and G.D. Cain, "A reliable control protocol for high-speed packet transmission", *IEEE Trans. on Commun.*, vol. COM-25, no. 10, pp. 1203-1210, October 1977.

[NERI84] Neri, G. et al., "MININET: a local area network for real-time instrumentation applications", *Computer Networks*, vol. 8, no. 2, pp. 107-132, April 1984.

[NESS79] Nessett, D., "A survey of congestion control issues in store-and-forward networks", Lawrence Livermore Lab. preprint UCRL-83551, November 1979.

[NIEM84] Niemegeers, I.G. and C.A. Vissers, "Twentenet, a LAN with message priorities, design and performance considerations", presented at the ACM SIGCOMM, Montreal, Canada, June 1984, published in *Computer Commun. Rev.*, vol. 14, no. 2, pp. 178-185.

[ORNS75] Ornstein, S.M. et al., "PLURIBUS - a reliable multiprocessor", AFIPS Conf. Proc., vol. 44, pp. 551-559, May 1975.

[PENN78] Penney, B.K. and A.A. Baghdadi, "Survey of computer communication loop networks", Research Report 78/42, Dept. of Computing and Control, Imperial College, London, September 1978.

[PETE72] Peterson, W.W. and E.J. Weldon Jr., *Error-Correcting Codes (2nd Edition)*, MIT Press, 1972.

[PIER72] Pierce, J.R., "Network for block switching of data", *Bell System Tech. J.*, no. 6, pp. 1133-1145, July 1972.

[PONC75] Poncet, F. and J.B. Tucker, "The design of the packet switched network for the EIN project", Proc. European Computing Congress (EUROCOMP-75), pp. 301-314, 1975.

[POUZ74] Pouzin, L., "CIGALE, the packet switching machine of the CYCLADES computer network", Proc. IFIP Congress (North Holland), Stockholm, Sweden, August 1974.

[PRIC77] Price, W.L., "Data network simulation: experiments at the National Physical Laboratory 1968-1976", *Computer Networks*, vol. 1, pp. 199-210, 1977.

[PROS62] Prosser, R.T., "Routing procedures in communication networks", *IRE Trans. Commun. Syst.*, vol. CS-10, pp. 322-335, December 1962.

[RAJA78] Rajaraman, A., "Routing in TYMNET", Proc. European Computing Congress (Eurocomp78), pp. 9-21, London, U.K., May 1978.

[RAUB76] Raubold, E. and J. Haenle, "A method of deadlock-free resource allocation and flow control in packet networks", Proc. 3rd Int. Conf. on Computer Commun., pp. 483-487, Toronto, Canada, August 1976.

[RAWS78] Rawsom E.G. and R.M. Metcalfe, "Fibernet: multimode optical fibers for local computer networks", *IEEE Trans. on Commun.*, vol. COM-26, no. 7, pp. 983-990, July 1978.

[REAM75] Reames, C.C. and M.T. Liu, "A loop network for simultaneous Transmission of variable-length messages", Proc. 2nd Annual Symp. on Computer Arch., pp. 7-12, Houston, Texas, January 1975.

[RIND76] Rinde, J., "TYMNET I: an alternative to packet technology", Proc. 3rd Int. Conf. on Computer Commun. (ICCC), pp. 268-273, Toronto, Canada, August 1976.

[RIND79] Rinde, J. and A. Caisse, "Passive flow control techniques for distributed networks", Int. Symp. on Flow Control in Computer Networks, session 5, paper 1, Paris, France, February 1979.

[RUDI76] Rudin, H., "On routing and 'delta routing': a taxonomy and performance comparison of techniques for packet switched networks", *IEEE Trans. on Commun.*, vol. COM-24, no. 1, pp. 43-59, January 1976.

[SALW83] Salwen, H.C., "In praise of ring architecture for local area networks", *Computer Design*, March 1983.

[SCAN69] Scantlebury, R., "A model for the local area of a data communication network - objectives and hardware organization", Proc. ACM Symp. on Data Communs., Pine Mountain, Georgia, 1969.

[SCHW80] Schwartz, M. and T.E. Stern, "Routing techiques used in computer communication networks", *IEEE Trans. on Commun.*, vol. COM-28, no. 4, pp. 539-552, April 1980.

[SHAR82] Sharpe, W.P. and A.R. Cash, Eds., *Cambridge Ring 82 Interface Specifications*, Science and Engineering Research Council, RAL Labs., Didcot, England, September 1982.

[SIND83] van Sinderen, M. and C.A. Vissers, "An architectural model for network interconnection", Proc. European Teleinformatics Conf. (EUTECO), pp. 475-488, Varese, Italy, October 1983.

[SLOM83] Sloman, M.S., R.C.S. Morling and D. Heger, "Activities of the local area network service specification group", Suppl. to Proc. European Teleinformatics Conf. (EUTECO), pp. 11-15, Varese, Italy, October 1983.

[STEW70] Steward, E.H., "A loop transmission system", Proc. 6th IEEE Conf. on Communs., pp. 36/1-36/9, San Francisco, California, June 1970.

[TARI83] Tarini, F. and P. Zini, "Channel access schemes for high-speed/long-distance LANs", Proc. European Teleinformatics Conf. (EUTECO), pp. 415-423, Varese, Italy, October 1983.

[THOR79] Thornton, J.E., "Overview of HYPERchannel", Proc. Computer Conf. (COMPCON), pp. 262-265, San Francisco, California, (IEEE) February 1979.

[TOKO77] Tokoro, M. and K. Tamaru, "Acknowledging Ethernet", IEEE Computer Commun. Conf. (COMPCOM), pp. 320-325, Fall 1977.

[TOUE79] Toueg, S. and J.D. Ullman, "Deadlock-free packet switching networks", Proc. ACM Symp. on the Theory of Computing, pp. 89-98, Atlanta, Georgia, 1979.

[TYME71] Tymes, L.R., "TYMNET - a terminal oriented communications network", Spring Joint Computer Conference, AFIPS Conf. Proc. vol. 38, pp. 211-216, Spring, 1971.

[VISS85] Vissers, C.A. and L. Logrippo, "The importance of the service concept in the design of data communication protocols", Proc. IFIP 6.1 5th Int. Workshop on Protocol Specification, Testing and Verification, pp. 3-19, Toulouse-Moissae, France, June 1985.

[WATS78] Watson. R.W., "The LLL Octopus network: some lessons and future directions", Proc. 3rd USA-Japan Computer Conf., pp. 12-21, San Francisco, California, 1978.

[WAWE82] Wawer, W., K. Emmelmann and V. Tachammer, "Ordered bus access by low level token passing for the local network TOPAS", Proc. 2nd European Symp. on Real-Time Data Handling and Process Control., pp. 237-239, Versailles, France, November 1982.

[WELL71] Weller, D.R., "A loop communication system for I/O to a small multi-user computer", Proc. 5th Annual IEEE Conf. on Hardware, Software, Firmware and Tradeoffs, pp. 49-50, Boston, Mass., September 1971.

[WILK75] Wilkes, M.V., "Communication using a digital ring", Proc. PACNET Conf., pp. 47-55, Sendai, Japan, August 1975.

LIST OF RELEVANT PUBLICATIONS

BY THE AUTHOR

The following papers have already been presented or published by the author and are directly relevant to the subject of this thesis.

"MININET: a real-time packet switching network", presented at Minicomputers in Data Communs., (PCL) Firenze, Italy, May 1974.

with G.D. Cain, "MININET: a packet-switching minicomputer network for real-time instrumentation", Proc. A.I.M. Int. Meeting on Minicomputers and Data Communs., Liege, Belgium, January 1975.

with G. Neri and G.D. Cain, "A reliable control protocol for high-speed packet transmission", *IEEE Trans. on Commun.*, vol. COM-25, no. 10, pp. 1203-1210, October 1977.

with G. Neri, G.D. Cain, E. Faldella, T. Salmon and D.J. Stedham, "The MININET internode control protocol", Proc. Symp. on Computer Network Protocols, pp. B4.1-B4.6, Liege, Belgium, February 1978.

with G.D. Cain, "MININET: A local area network for real-time instrumentation and control", Proc. 3rd Conf. on Local Computer Networking, Minneapolis, Minnesota, October 1978.

with G.D. Cain, T. Salmon-Cinotti, E. Faldella, G. Neri and P.M. Stevens, "The value of transparency in local area data networks", Proc. 1st European Symp. on Real-Time Data Handling and Process Control, pp. 691-696, Berlin, Germany, October 1979.

with G.D. Cain, G. Neri and M. Longhi-Gelati, "MININET: an ultra-transparent local area network for service to high-speed instrumentation users", presented at Distributed Computing: a Review for Industry, Manchester, U.K., March 1983.

with G.D. Cain, "A routing protocol that maintains packet sequency", Proc. Mediterranean Electrotechnical Conf. (MELECON '83), vol. 1, paper A3.03, Athens, Greece, May 1983.

"DIM: a network compatible intermediate interface standard", *Interfaces in Computing*, vol. 1, no. 2, pp. 117-144, 1983.

with M.S. Sloman and D. Heger, "Activities of the local area network service specification group", Suppl. to Proc. European Teleinformatics Conf. (EUTECO), pp. 11-15, Varese, Italy, October 1983.

with G. Neri, G.D. Cain, E. Faldella, M. Longhi-Gelati, T. Salmon-Cinotti and P. Natali, "MININET: a local area network for real-time instrumentation applications", *Computer Networks*, vol. 8, no. 2, pp. 107-132, April 1984.

"System controller design", *Electronic Product Design*, Vol. 6, Part 1: no. 7, pp.55-58, July 1985; Part 2: no. 8, pp. 45-48, August 1985; Part 3: no. 9, pp. 81-83, September 1985.

with A.J. Carter, "Hierarchical system design and the engineering workstation revolution", presented at the Int. Computer Graphics User Conf., London, U.K., February 1985.