An Ethernet Bridge with Packet Filtering and Statistics Collection Capabilities

by

Amir Amanullah Khan

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

July, 1996

An Ethernet Bridge With Packet Filtering And Statistics Collection Capabilities

BY

Amir Amanullah Khan

A Thesis Presented to the
FACULTY OF THE COLLEGE OF GRADUATE STUDIES
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In

Computer Engineering

July 1996

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN-31261, SAUDI ARABIA.

COLLEGE OF GRADUATE STUDIES

This thesis, written by **Amir Amanullah Khan** under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

Thesis Committee

Dr. Muhammad S. T. Benten (*Thesis Advisor*)

Dr. Naser Al-Darwish (*Co-Chairman*)

Dr. Mohsen Guizani (*Member*)

Dr. Habib Youssef (*Member*)

Dr. Muhammad S. T. Benten
(*Department Chairman*)

Dr. Ala H. Al-Rabeh
(*Dean, College of Graduate Studies*)

17·6·96
Date

Dedicated to

my parents

بسم الله الرحمن الرحيم

Acknowledgment

I wish to thank my thesis committee chairman Dr. M. S. T. Benten, and co-chairman Dr. Nasir Al-Darwish for their continued guidance, and committee members Dr. Mohsen Guizani and Dr. Habib Youssef for their support.

Thanks are due to the acting department chairman, Dr. M. S. T Benten for his commitment to lesser work load during exam times. I also wish to thank Dr. Habib Youssef for having the time for me whenever I required it. Acknowledgment is due to King Fahd University of Petroleum and Minerals for allowing me to continue as a part-time student while serving as a Lab Engineer in my department, without which this work would not have been possible.

My thanks to the graduate students of the departments of Computer Engineering and Information & Computer Sciences and my friends especially Asjad Mumtaz from whom I learned a lot and who made the long work hours pleasant.

Lastly, but not the least, I wish to thank my family for their patience and endurance during the long hours while I was working on my thesis.

Contents

123

List of Tables

List of Figures

123

123

Abstract

Bridges provide a means to extend the LAN environment in physical extent, number of stations, performance, and reliability. They also provide a means to interconnect dissimilar LANs. A bridge works on the data link layer, and transfers only those frames from one segment to the other that have destination addresses on the other segment. Security and Network monitoring are two very important concerns when designing and maintaining networks.

In this thesis a two port transparent Ethernet bridge has been designed and implemented. Its behavior and performance have been tested and compared to other similar work. The bridge was written to conform to the Packet Driver Specification Standard, making it hardware independent. The software was implemented in the C programming language and runs on a standard PC with two Ethernet network interface cards. The bridge provides two additional functions: packet filtering and statistics collection. Packet filtering is used to build security firewalls, specifically packet filter gateways. Statistics collection has diverse applications and is used in accounting, design of protocols, network expansion and developing network subsystems. Statistics is collected and periodically sent, in a statistics datagram, to a central location for further processing and a global perspective of the complete network.

الخلاصة

توفر الجسور طريقة لتوسعة شبكات الحاسب الآلي المحلية (LAN) من النواحي الطبيعية و عدد محطات العمل و الاداء والكفاءة. كذلك تمثل الجسور طريقة لربط الشبكات الغير متماثلة. ويعمل الجسر في طبقة ربط المعلومات حيث يقوم بنقل الاطارات من قطاع الى آخر حيث عنوان الوصول. ويمثل الامن ومراقبة الشبكة عنصران هامان عند تصميم الشبكة وكذلك الحفاظ عليها.

ونعرض في هذه الاطروحة تصميم وتنفيذ جسر خاص بشبكة (ETHERNET) ذو طرفي ربط. وقد تم تصميم هذا الجسر ليتوافق مع توصيف مدور الحزم (Packet Driver) القياسي وكذلك ليكون غير معتمد على البناء الطبيعي للشبكة. ولقد تم تنفيذ هذا الجسر بلغة البرمجة سي وتشغيله على جهاز حاسب آلي متوافق مع كارتي توصيل لشبكة (ETHERNET). ويتميز هذا الجسر بميزتان اضافيتان وهما القدرة على تنقيح الحزم وجمع الاحصائيات. ويتم الاستفادة من خاصية تنقيح الحزم في بناء جدار أمني خاصة لمنافذ الشبكة. ويتم الاستفادة من عملية جمع الاحصائيات في عدة تطبيقات مثل المحاسبة وتصميم البروتوكولات وتوسيع الشبكة. ويتم جمع الاحصائيات وارسالها بطريقة دورية في صورة بيانية الى الوحدة المركزية لجمع البيانات من كافة اجزاء الشبكة ثم الاستفادة منها.

CHAPTER 1

1 INTRODUCTION

A computer communication network is an interconnection of computers that uses a communication channel, such as an electrical cable, radio waves or light beam etc., to share resources like data, CPU and peripheral devices.

Bridges provide a means to extend the LAN (local area network) environment in physical extent, number of stations, performance, and reliability. In some cases they are also required to provide a means to interconnect dissimilar LANs. Apart from extending the electrical limitations on node number and cable length, a bridge transfers only those frames from one segment to another that have non local destination addresses (destination and source addresses are in separate segments). The bridged LAN should retain as many properties of a single LAN as possible. This includes high throughput, low delay and a low occurrence of frame loss, misordering or duplication. They are also required to preserve the general broadcast nature of the channel [1].

Another issue related to networks is monitoring the network traffic for protocol performance and service usage. This information is used to plan and design the network for expansion, and also in improving its throughput and utilization. Traffic parameters like frame lengths and their distributions (which may have been filtered on protocol or service), are required during the design and testing of new protocols and network subsystems. Traffic statistics are also used for accounting purposes, to charge customers according to usage.

As more and more networks get connected together the issue of computer and data security becomes a real concern. This gives rise to the concept of firewalls. Networks need firewalls for their protection. Firewalls allow only certain legitimate network addresses to connect to the network, and/or limit and control the services provided by the network.

In this thesis we are concerned with the design and software implementation of a two port transparent Ethernet bridge for the DOS platform. Standard PC hardware and network interfaces are to be used. The bridge is to conform to the Packet Driver interface specification. The bridge should perform the additional functions of packet filtering and statistics collection.

1.1 Motivation

Commercial bridges are generally implemented in hardware for performance reasons. Software implementations yield lower performance because software is inherently slower than hardware. On the other hand, software implementations provide ease of enhancements and modifications. Recently some work has been done in this field at Ohio State University, where the "KarlBridge" program has been developed [2]. KarlBridge is a software bridge, written for the PC platform, and uses a particular model of network interface cards from a particular vendor (SMC Elite 8 and Elite 16 cards with WD83C690 chip). It uses custom network interface drivers (for better performance) which makes it difficult to port to other hardware. It also makes it harder to adapt to newer technology.

With today's fast hardware technology it is felt that a bridge written to some interface specification standard would provide reasonable performance. By conforming to standards, such a bridge will be able to use the latest technology without any modifications. It will also be able to exploit the best technology available from the various vendors. Another advantage of following standards is that the functions developed for the bridge, such as initializing the network interfaces, receiving frames, sending frames and filtering, can be used in the development of a large number of network subsystems like Internet firewalls, network monitors, higher level gateways, etc. This is especially true for applications that do not demand a very high forwarding rate, like Internet firewalls. Security firewalls are generally placed before out-going wide area network (WAN) links. WAN links are generally much slower than LANs. (T1 and E1 carriers at 1.544 and 2 Mbps are popular choices for WAN links). A software bridge, with relatively low throughput as compared to a hardware bridge, will not become a bottleneck if placed before a WAN link. And therefore it provides a reasonable platform to implement a security firewall.

Other motivating factors were to learn about intricacies and interactions of network protocols at the basic frame level and to develop and implement a real time application that must meet high performance standards.

1.2 Thesis Organization

In this thesis a transparent Ethernet bridge was designed and implemented, using standard PC hardware and software. IEEE 802.1 part D recommendations, for developing LAN bridges were followed [3]. A public domain interface specification standard, namely the Packet Driver Specification [4], was followed to ensure that the bridge is hardware independent. The bridge also provides additional functions of packet filtering and statistics collection. It discards frames on the basis of *IP* (Internet Protocol) addresses and other configurable parameters, like protocol or service . It collects local segment traffic statistics and sends the statistics bearing datagram to a central location for further processing. The central location can process the local statistics from various segments and develop a global view of the whole network.

Chapters two and three provide the necessary background for understanding the design of bridges and the packet filtering operation. Chapter two gives a description of the different types of frame formats used on Ethernet. Both IEEE 802 and de-facto standards are presented along with the differences between them. Chapter two also introduces the various interface specification standards prevalent in the industry. Chapter three describes IEEE 802.1 recommendations for MAC (Medium Access Control) level bridges. It also presents the various types of bridges, along with their basic design issues like buffering strategies and database look-up principles. It then presents the notion of network security and firewalls, and discusses various strategies to implement them. The chapter also introduces the concept of local traffic monitoring and building a global perspective of the network traffic.

Chapter four is mainly concerned with the design and implementation of the bridge and its two additional functions (packet filtering and statistics collection). It describes the various design issues, the choices available, the approach taken, and the reasons for taking that approach.

The proposed bridge was tested and compared with other similar bridges. Chapter five presents the experimental evaluation of the proposed bridge. It describes the test set-up used, limitations of the test set-up, and the main differences and similarities between the proposed bridge and other reported bridges. Experimental results are then presented and discussed. In this chapter, we also identify some resource shortcomings which affect the bridge's performance adversely and suggest some measures that may be taken to further improve the bridge performance.

Conclusions and suggestions for future work are given in chapter six. This chapter also presents some of the non-conventional applications for which the bridge, as developed, may be used.

1 .3 Source Code

The source code of the proposed bridge and the "statistics display module" are stored on a diskette and attached on the back cover of the thesis. A set of executables and packet drivers for some common interface cards are also provided.

1 .4 Conclusions

In this chapter the problem of designing an Ethernet bridge with two additional functions of packet filtering and statistics collection was formulated. The motivations for taking up this project were also discussed. In the next chapter ISO and IEEE communication models and the frame formats used on Ethernet are presented. De-facto interface specification standards for the DOS and Windows platforms are also presented.

CHAPTER 2

2 LAN ARCHITECTURE AND PROTOCOLS

This Chapter provides the necessary background for understanding the functionality and design of bridges, and packet filters. It gives a summary of the proposed models for communication and discusses the different frame formats used in Ethernet. It also introduces the various interface specification standards prevalent in the industry and gives a summary of the Packet Driver specification.

2.1 Communication Models

Two international standardizing bodies ISO (for WANs) and IEEE (for LANs) have proposed computer network architecture models for the communicating bodies. ISO has proposed a seven layer model in which each layer has a well defined function [5]. The communicating entity representing a layer uses a protocol to communicate with its peer entity on the distant machine. A *protocol* is a well defined set of rules to carry out conversation in an orderly structured manner between peer processes, running on the communicating computers. Adjacent layers have a well defined *interface* between them which interprets formats and other protocol features from one protocol layer to another (see Figure 2-1). Both the ISO and IEEE models are summarized in the following sections.

2.1.1 The OSI Reference Model

ISO has defined a seven layer reference model for data networks. This is called the OSI (Open Systems Interconnection) reference model. The seven layers of the model are Physical Layer, Data Link Layer (DLL), Network Layer (NL), Transport Layer (TL), Session Layer (SL), Presentation Layer (PL) and Application Layer (AL) [5]. The OSI reference model is summarized in Figure 2-2.

2.1.2 IEEE 802 Project

The Institute of Electrical and Electronics Engineers (IEEE) undertook project 802 in order to standardize LAN protocols. The project defines three sublayers, Logical Link Control (LLC), Medium Access Control (MAC) and Physical Layer. IEEE 802.1 is a document that describes the relationship between the various parts of the standard. IEEE 802.2 describes the logical link control (LLC) protocol and is

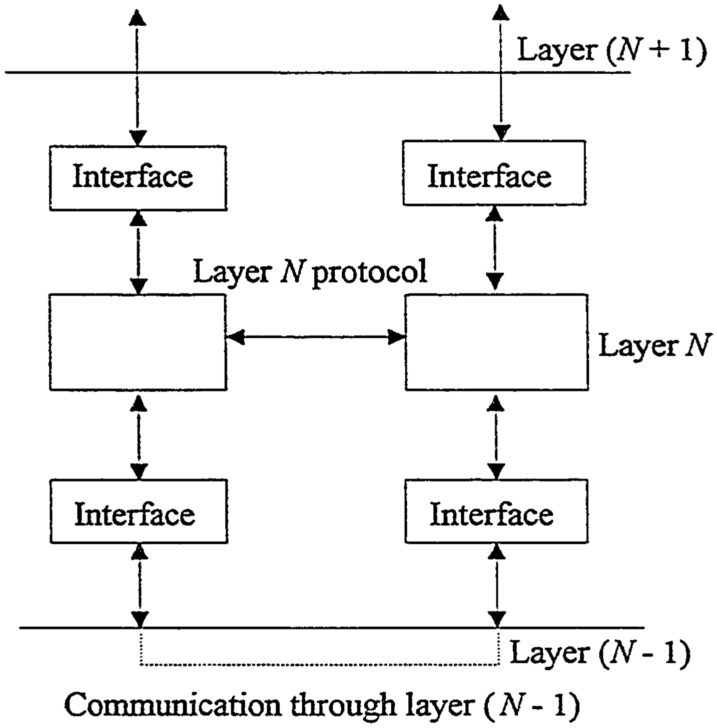


Figure 2-1 Protocol and Interface

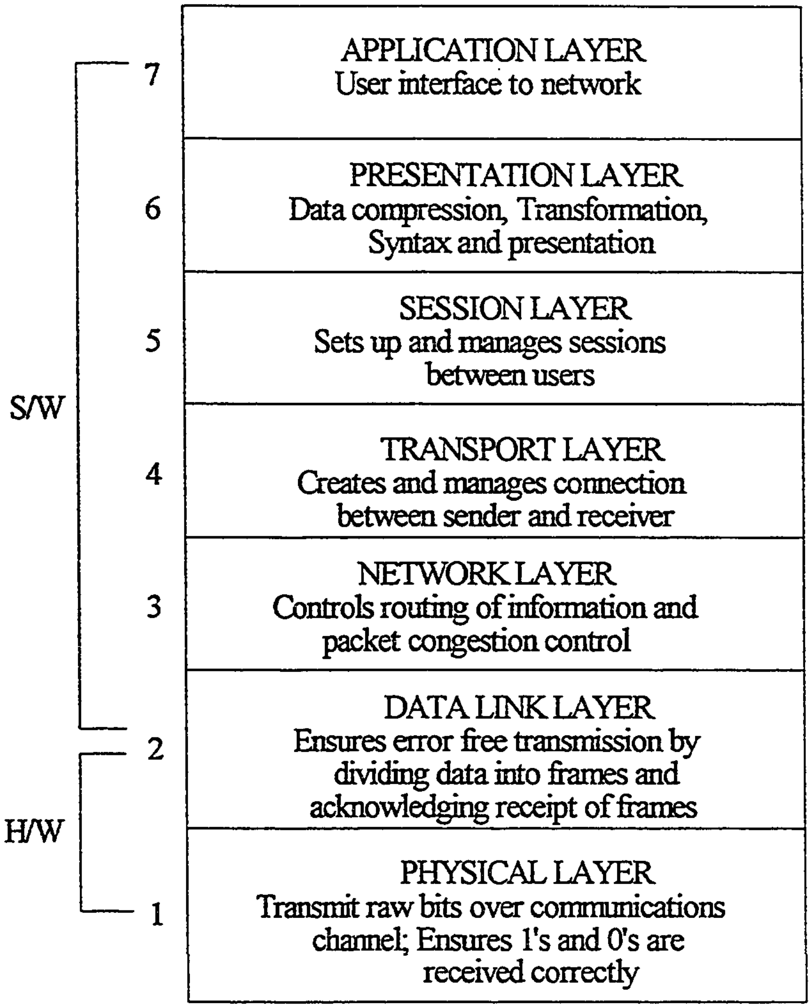

APPLICATION LAYER
User interface to network
7
PRESENTATION LAYER
Data compression, Transformation, Syntax and presentation
6
SESSION LAYER
Sets up and manages sessions between users
5
TRANSPORT LAYER
Creates and manages connection between sender and receiver
4
NETWORK LAYER
Controls routing of information and packet congestion control
3
DATA LINK LAYER
Ensures error free transmission by dividing data into frames and acknowledging receipt of frames
2
PHYSICAL LAYER
Transmit raw bits over communications channel; Ensures 1's and 0's are received correctly
1
S/W
H/W

Figure 2-2 OSI Reference Model

responsible for multiplexing multiple channels to and from a single physical interface. IEEE 802.X (X = 3, 4, 5) is the Medium Access Control (MAC) standard [5] (see Figure 2-3). A brief description of the IEEE standards follows.

2.1.2.1 LLC : IEEE 802.2.

The LLC performs some of the functions of ISO's DLL and is also responsible for some network layer functions. It is responsible for the reliable delivery of frames between adjacent stations. It is also responsible for error correction and flow control. Since all stations share a common medium, the network layer functions of routing and switching are not required. A major difference between the OSI's DLL and LLC is that LLC provides for multiple end points to the datalink. Multiple logical links can be maintained using SAPs (Service Access Points). The LLC header provides for the addressing of both the destination (DSAP) and source (SSAP) service access points [6, 5] (see Figure 2-4). Three types of services are provided as outlined below:

(i) Type 1 (Unacknowledged Connectionless) Service: This is a datagram service which supports point-to-point, multipoint, and broadcast modes.

(ii) Type 2 (Connection Oriented) Service: Provides logical connections between SAPs; it provides flow control, error control, and sequence control.

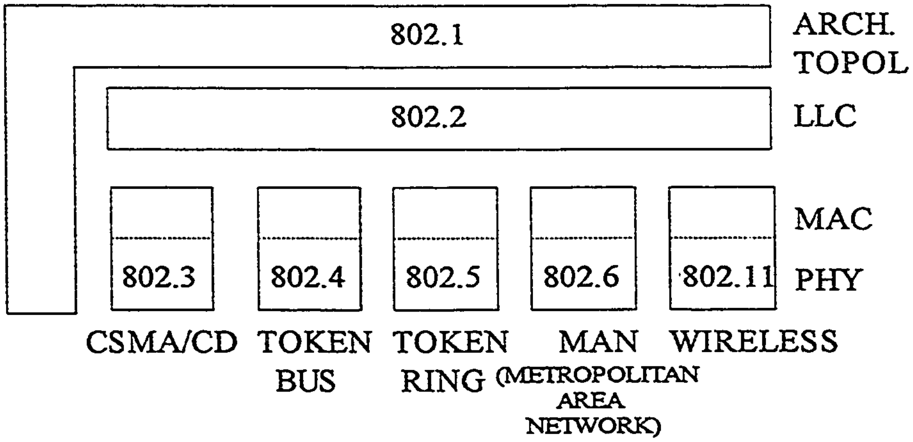


Figure 2-3 IEEE 802 Project

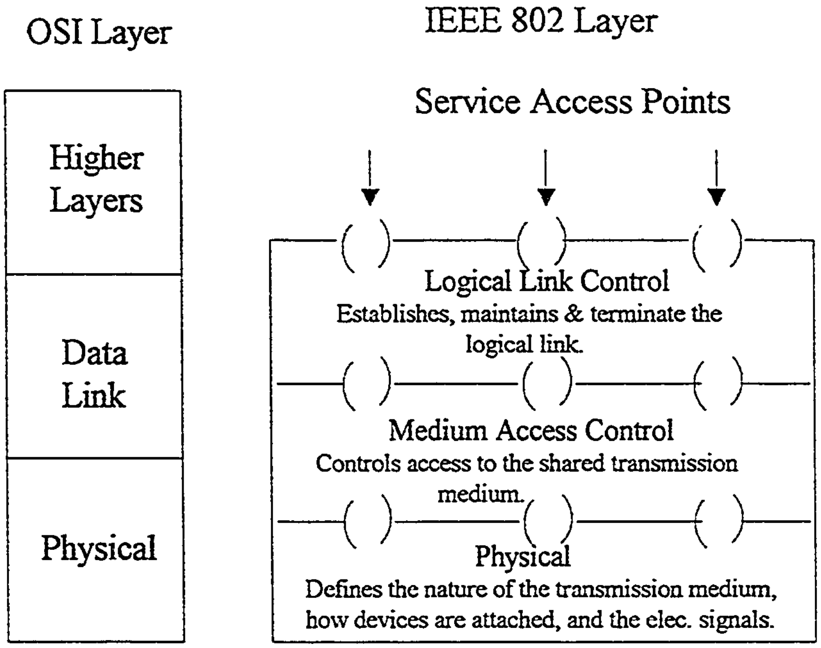

OSI Layer
IEEE 802 Layer
Service Access Points
Higher Layers
Data Link
Physical
Logical Link Control
Establishes, maintains & terminate the logical link.
Medium Access Control
Controls access to the shared transmission medium.
Physical
Defines the nature of the transmission medium, how devices are attached, and the elec. signals.

Figure 2-4 Relationship Between OSI and IEEE 802

(iii) Type 3 (Acknowledged Connectionless) Service: This is also a datagram service with point-to-point links and acknowledgments.

The LLC services as defined above do not provide for IP (Internet Protocol) encapsulation in the ethernet frame. IP is an integral part of TCP/IP, which is a de-facto standard for the Internet. To allow IP encapsulation the Sub-Network Access Protocol (SNAP) was introduced.

Ethernet SNAP: The Sub-Network Access Protocol (SNAP) has been defined as an extension of the LLC to allow encapsulation of *IP* (Internet Protocol) datagrams and *ARP* (Address Resolution Protocol) requests and replies within an 802.X (X=3,4,5) frame.

2.1.2.2 MAC : IEEE 802.X (X = 3, 4, 5)

The MAC overlaps the physical and datalink layers of the OSI model and provides for the use of random access or token passing procedures for controlling the access to the medium (see Figure 2-4). Framing, addressing and error detection fall in the domain of the MAC sublayer. The IEEE 802 project has specified three different MAC frame format standards for LANs, these are:

(i) IEEE 802.3 describes the frame format to be used with CSMA/CD on a bus.

(ii) IEEE 802.4 describes a token passing scheme over a bus.

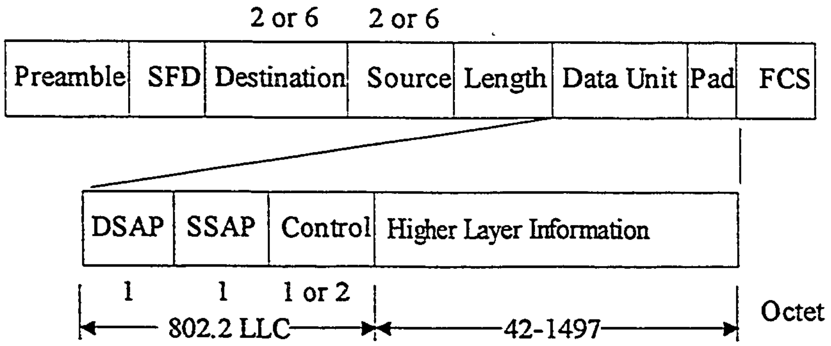
(iii) IEEE 802.5 describes the frame formats for token ring networks; the format of the token, information frame and abort sequence are specified.

2.2 Frame Formats

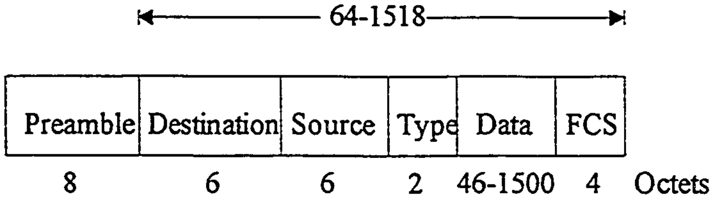
In practice Ethernet uses two basic frame types. One that was introduced before the standardization process and has become de-facto standard for certain protocols, like TCP/IP (Transmission Control Protocol / Internet Protocol), and the other that conforms to the IEEE 802 standard [6, 7]. The frame types are described below.

2.2.1 IEEE 802.3 Frame Format

This standard has already been discussed in the previous section. The format of the frame is shown in Figure 2-5. The frame starts with a frame start delimiter (preamble and SFD) followed by two or six octet long destination or source address fields. The next two octets contain the total length of the frame in octets. If there is no LLC header then the next field is the data carried by the frame. This is a variable length field with a fixed minimum and maximum length. The minimum length for data is forty six octets. If the data is less than the minimum forty six octets then padding is used to bring it to this value. In the end of the frame is its frame check sequence (FCS), which is required for error checking.



Ethernet 802.3



Ethernet DIX

Figure 2-5 IEEE 802.3 and DIX Frame Formats

2.2.2 De-Facto Frame Standard

Apart from ISO and IEEE 802 standards there is an important frame format for Ethernet used in practice. This is called DIX Ethernet, Ethernet II, or Blue Book Ethernet. This Ethernet frame format was developed by DEC, Intel, and Xerox (DIX) and is slightly different than the IEEE 802.3 format. The main differences are [6, 7]:

(i) The source and destination addresses are strictly six octets for DIX while they may be two or six octets for IEEE 802.3, although six octets is more common.

(ii) The Type field of DIX is replaced by the length field of 802.3. Type signifies protocol type, e.g. $(0800)_{HEX}$ for IP, while the length field indicates the number of LLC octets in the data field.

(iii) DIX has no provision to pad data to its minimum of 46 octets, but 802.3 does. Hence the upper layers take care of frame size validity for DIX.

Frame formats for IEEE 802.3 and Ethernet DIX are shown in Figure 2-5.

2.3 Multiple Protocol Stacks

Some applications, such as bridges, need to receive / send frames belonging to different protocols on a single network interface card, at the same time. To achieve this, interface specification standards are formulated. These are documents that describe the service primitives to be used while communicating between the two adjacent layers. Applications and drivers are both written conforming to the standard. The standard also defines the method and procedure of passing information between the two entities. Thus, the application may be written without any knowledge of the internal workings of the driver. Three different interface specification standards have been developed for DOS and Windows, by various companies [8, 9]. Nearly all the major protocols, on top of all the major MAC standards (i.e. IEEE 802.3, IEEE 802.4 and IEEE 802.5) are supported by all of them. The three interface standards for protocol stacking are NDIS (Network Device Interface Specification), ODI (Open Data Link Interface), and Packet Drivers (see Figure 2-6). All of the standards provide device independence by defining procedures and function calls, along with their associated parameters, to control the network interface card. They also to provide a standard interface to the application. A brief description of these standards follows.

2 .3.1 NDIS (Network Device Interface Specification)

Microsoft and 3Com have proposed the NDIS specification. It conforms to the MAC of the IEEE model and has a protocol manager with a vector module. When a data frame comes, the vector module asks each loaded protocol stack if the frame is of that protocol type. If it is, the protocol stack accepts it, otherwise, the frame passes on to the next protocol stack.

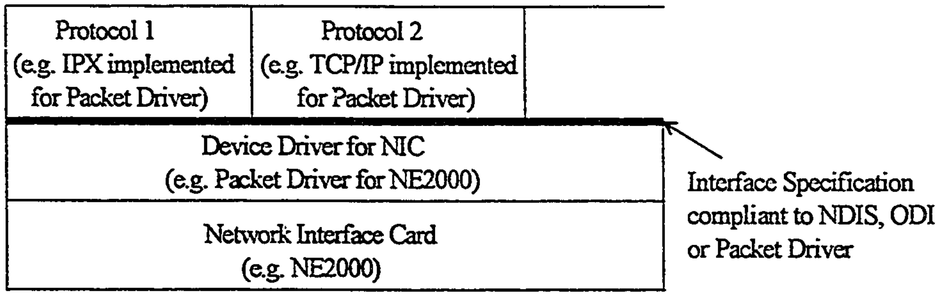

Protocol 1
(e.g. IPX implemented for Packet Driver)
Protocol 2
(e.g. TCP/IP implemented for Packet Driver)
Device Driver for NIC
(e.g. Packet Driver for NE2000)
Network Interface Card
(e.g. NE2000)
Interface Specification compliant to NDIS, ODI or Packet Driver

Figure 2-6 Multiple Protocol Stacks

All major LAN protocols have been implemented using the NDIS specification. Another specification, WANDIS (Wide Area Network Device Interface Specification), has been proposed for wide area connectivity for protocols like X.25.

2.3.2 ODI (Open Data Link Interface)

This interface specification is supported by Novell and Apple. ODI has three parts: the Multiple Protocol Interface (MPI), which communicates with the protocol stacks, the Multiple Link Interface (MLI), which communicates with the driver of the adapter card, and the Link Support Layer (LSL), which links communications between MPI and MLI. When a data frame arrives, the LSL determines to which stack it should go. The price paid for multiple protocols is a slight speed degradation, which can be 5% slower than monolithic protocols.

2.3.3 Packet Driver

FTP Inc. first developed the packet driver specification and later released it to the public domain. Unlike the ODI and NDIS, the packet driver does not build the Ethernet headers. It leaves this job to the application. Major protocols like TCP/IP, IPX, NetBIOS, Apple Talk, have been implemented using the packet driver specification [4].

The packet driver specification revolves around a software interrupt which is invoked to perform a variety of functions like transmitting / receiving frames, getting information about the network interface, retrieving statistics from the interface, etc. The application first registers itself with the packet driver and informs it about the mode it is going to use. The mode determines what frames are going to be passed to the application, e.g. whether only frames destined to itself are to be passed on, or all frames are to be passed. On receipt of a frame, an interrupt is generated which requests buffer space for the frame, the frame is copied into the buffer which is then taken in charge by the application.

For illustration, the specification of a function supported by the Packet Driver specification and its interpretation are given. The function is called *send_pkt()* and is used to transmit frames. The specification is given below:

send_pkt()

int send_pkt(buffer, length) *AH == 4*

*char far *buffer;* *DS:SI*

unsigned length; *CX*

error return:

carry flag set

error code *DH*

possible errors:

CANT_SEND

non-error return:

carry flag clear

Transmits length bytes of data, starting at buffer. The application must supply the entire frame, including local network headers. Any MAC or LLC information in use for frame demultiplexing (e.g. the DEC-Intel-Xerox Ethertype) must be filled in by the application as well. This cannot be performed by the driver, as no handle is specified in a call to the send_packet() function.

The specification for the *send_pkt ()* function describes how to transmit a frame. The contents of a buffer, whose address is contained in the *data segment* and *source index* registers of the CPU, are transmitted when the software interrupt of the interface driver is invoked, after placing a value of 4 in the *AH* register. The *data segment* register should contain the segment address of the buffer and the *source index* should have its offset. The length of the buffer should be placed in the *CX* register before invoking the interrupt. The application (or upper layer) using the driver is responsible for forming the complete frame, including the physical layer headers. Only the frame's checksum is appended by the interface. If there is a transmission error, the carry flag is set on the return of the interrupt otherwise it is clear. The error code is returned in the *DX* register. The error codes are given in appendix A4. The C language code has been written to perform these functions.

2.4 Summary Of The Packet Driver Specification

There are three types of packet driver calls: basic, extended, and high performance. All packet drivers support basic calls. Some support extended or high performance calls as well, while few support all the three types. Following is a summary of the packet driver calls [4]. *(E)* indicates an extended function and *(H)* indicates a high performance function:

driver_info

Indicates which packet driver calls (i.e. basic, extended, or high performance) are supported.

access_type

The application calls the access_type function to register itself with the packet driver and inform it of the interrupt handler to call when a frame shows up.

release_type

Used to unhook the interrupt handler from the packet driver.

send_packet

To send the frame in the buffer, whose address and length are passed to the call.

terminate

Unloads the packet driver.

get_address

Get the interface Ethernet address.

reset_interface

Reinitialize the network interface.

get_parameters (H)

Get information about the major and minor version numbers of the packet driver and also the hardware.

set_rcv_mode (E)

Six receive modes are possible. Mode 1 turns off the receiver. Mode 2 receives only directly addressed frames. Mode 3 receives directly addressed and broadcast frames. Mode 4 receives directly addressed and broadcast frames and limited Ethernet multicast addresses. Mode 5 receives directly addressed and broadcast frames and all multicast addresses. There is a difference between the multicast addresses supported in mode 4 and mode 5. In mode 4 the multicast addresses supported are set in the network interface while in mode 5 they are set in both the network interface and the network interface driver. Mode 6 receives all frames (promiscuous mode).

get_rcv_mode (E)

Returns the current receive mode.

set_multicast_list (E)

Set the hardware multicast list.

get_multicast_list (E)

Get the current hardware multicast list.

get_statistics (E)

Allows reading of hardware or packet driver generated statistics.

set_address (E)

Set the interface Ethernet address.

2.5 Conclusions

In this chapter the ISO and IEEE communication models and the frame formats used on Ethernet were studied. These are required for understanding bridging and packet filtering operations. Interface specification standards for DOS, which define the interface between the application and network interface driver, were also presented. The next chapter presents the IEEE recommendations for designing bridges and summarizes their types. It also presents the design issues of bridges. Finally, it introduces the additional functions of packet filtering and statistics collection, along with their need and use.

CHAPTER 3

3 LAN BRIDGING CONCEPTS

This chapter provides the background for bridge design, and introduces the additional functions of *packet filtering* and *statistics collection.* First, it presents IEEE 802 recommendations for LAN bridge design. Then it discusses other classifications of bridges based on implementation, topology supported, added functionality etc. Next the major bridge design issues are covered. Finally, the two additionally implemented functions of *packet filtering* and *statistics collection* are presented along with their need and use. Packet filtering has been covered mainly from the perspective of an Internet firewall.

3 .1 IEEE Recommendations For LAN Bridges

Bridges are devices that are used to extend the electrical and DLC protocol limits of a network. As mentioned earlier, they interconnect two or more LANs (either similar or dissimilar) at the media access level (MAC) of the datalink layer (see Figure 3-1). In order to maintain LAN characteristics on LANs which have been extended by bridges, IEEE 802 committee developed two standards for bridges. Part D of IEEE 802.1 defines bridge standards for connecting LAN segments based on any of the IEEE 802 MAC specifications. For Ethernet (IEEE 802.3) a transparent bridge is suggested. The transparent bridge can use the spanning tree algorithm to prevent frame loops from forming. Loops can be formed by closed paths in the bridged LAN. Their effect is to continuously circulate the frame in the loop, thereby adding unnecessary traffic. The second standard uses source routing approach and is recommended for interconnecting token-ring LANs (IEEE 802.5) [3, 10, 1]. The two recommended approaches are summarized below.

3 .1.1 Transparent Bridges

Transparent bridges are called “transparent” because the source and destination nodes are unaware of their existence, i.e. no modifications are required in them in order

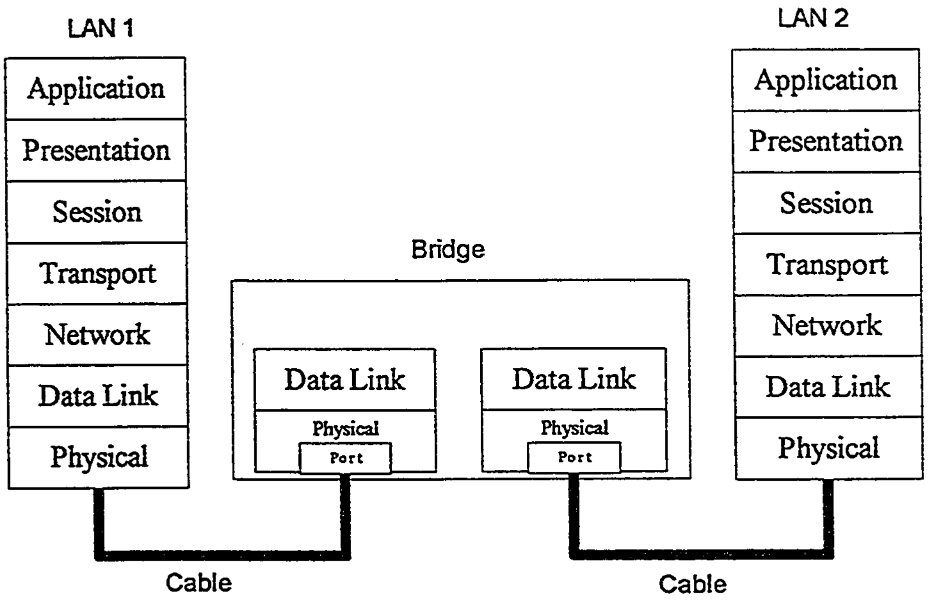


Figure 3-1 Bridges between two LANs function at the data link layer

to install the bridge. In general bridges perform a filtering function based on destination Ethernet addresses. Only valid MAC frames originating in one network segment and destined for the other network segment are transmitted over the bridge. The bridge continually checks the source address of all frames arriving from each interface, and uses these addresses to construct a routing / forwarding table for that particular interface. When a frame arrives, its destination address is compared with the addresses in each interface table of addresses. If the address is contained in the address table of the source interface, then the bridge does nothing because the destination address is located in the source segment. Otherwise, the frame is transmitted over the interface whose address table contains the destination address. This means that the traffic generated by a node on one side of the bridge and meant for another node on the same side is not passed onto the other side of the bridge. This effectively helps in alleviating congestion and hence contributes in improving efficiency / throughput of the network. The filtering capability also enhances the systems security by disallowing unwanted traffic from flowing to other parts of the network.

At the start, when address tables are empty, the frames are transmitted over all segments except the source segment. To allow for addition / removal of computers from segments, and other failures, each address entry has a timer associated with it. The timer is restarted whenever the bridge receives a frame from that address. If the timer times out before being restarted, then that entry is removed. The logical flow of the forwarding and learning algorithm is shown in Figure 3-2.

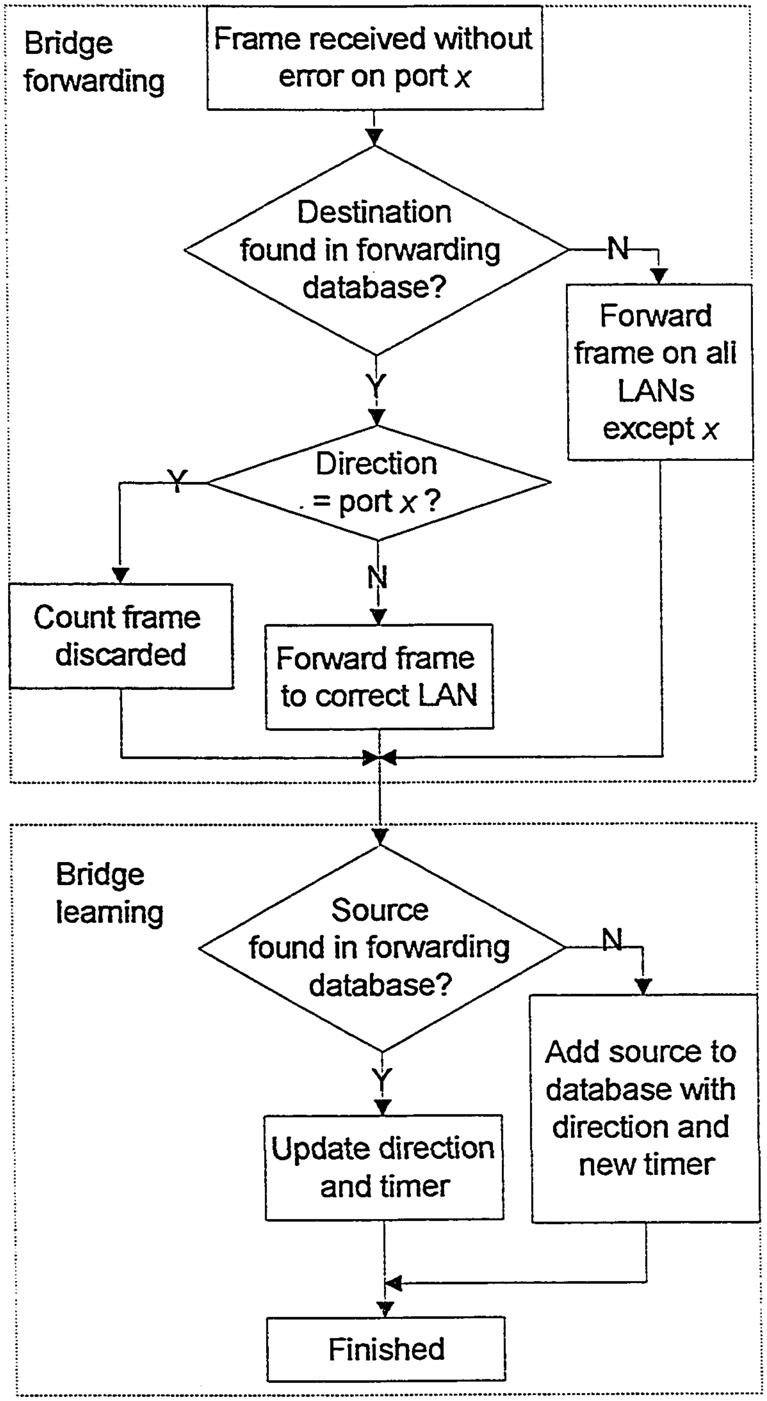

Bridge forwarding
Frame received without error on port x
Destination found in forwarding database?
N
Forward frame on all LANs except x
Y
Direction = port x ?
Y
Count frame discarded
N
Forward frame to correct LAN
Bridge learning
Source found in forwarding database?
N
Add source to database with direction and new timer
Y
Update direction and timer
Finished

Figure 3-2 Logical flow for bridge forwarding and bridge learning in a transparent bridge

3.1.1.1 Spanning Tree Algorithm

When the network contains more than one bridge, it is usually the case that there is more than one path between two LAN segments. This may result in frame loops because the learning process gets confused about the location of an end station, since the same source address can come in at several ports of the bridge. In 802.3 LANs the spanning tree algorithm is used to ensure that there is only one path between any two segments of the entire bridged LAN. Each bridge is given a unique bridge identifier and each of its ports is also given a unique port identifier. To form the spanning tree, first a root bridge is selected. Each bridge designates a root port within itself, which is the port with the least cost to the root bridge. For each LAN, a specific designated bridge is then chosen. Finally, in each designated bridge all designated ports as well as the root port are put into forwarding state while all the other ports are put in blocked state.

The root bridge sends "*hello*" messages periodically, using bridge protocol data units (BPDUs). Each time a designated bridge receives a BPDU, from the root bridge on its root port, it sends out a response BPDU through all of its designated ports. If a designated bridge were to fail, it would stop sending response BPDUs to the root bridge. If a redundant bridge is present, it would be requested to replace the failed bridge. This scheme allows for topology changes in case of bridge failure or addition.

3 .1.2 Source Routing Bridges

In source routing, the source specifies the complete path followed by the frame to reach its destination. Although source routing can be used for any type of LAN e.g. token bus or CSMA/CD, it has been standardized by IEEE 802 committee for IEEE 802.5 LANs. The algorithm is briefly described below:

The source sends out a *discovery* frame which floods the entire bridged LAN. Thus these frames travel through all possible paths between source and destination. On the way, each frame records the route it takes and, upon reaching the destination all the frames are returned back to the source along the same recorded paths. The source then chooses the route it is going to use. Thus, source routing bridges add source route information to the frame, whereas transparent bridges do not modify the frame.

3 .2 Other Classifications Of Bridges

There are various other classifications of bridges found in the literature, which are on the basis of some attribute like implementation, topology, or functionality [11, 12]. These classifications are not widely adopted and are summarized here for completeness.

1. On The Basis Of Implementation:

a) *Hardware Vs Software*: Hardware bridges are implemented using microprocessors, ROM code, and the ports required to connect to the network(s). They perform a lot faster than software bridges but generally support lesser functions due to cost. On the other hand *Software bridges* generally have more configurable options. Hardware bridges are dedicated bridges and may even start forwarding the frame onto the correct destination port before completely receiving it, while software bridges first buffer the frame and then retransmit it over the correct port.

b) *Switching Bridges*: Switching bridges use a physical switch internally rather than a software algorithm. They generally support the concept of virtual LANs as well.

2. On The Basis Of Topology:

a) *Local Vs Remote*: Another classification is whether the bridge is *local* or *remote*. A local bridge is a device which is used to connect two (or more) networks in close proximity and attaches to both networks physically. A remote bridge is used to connect two networks over a long distance, using a WAN link. In this case two bridges are required, one at each end, to do protocol conversion from Ethernet to the WAN protocol (e.g. X.25) at one end and from the WAN protocol back to Ethernet on the other end. In practice the whole Ethernet frame is encapsulated in a WAN frame and transmitted over to the other end, where the WAN frame is stripped off. Thus, the two networks share the same broadcast channel despite the distance.

3. <u>On The Basis Of Functionality:</u>

a) *Translation Bridges :* Some bridges have the capability to connect dissimilar MAC technologies such as connecting Ethernet to token ring. These are special bridges called *translation bridges* and are required to perform additional functions of frame format translation. Apart from frame format translation, the bridge may also be required to perform other functions to harmonize the two different MAC technologies. For instance, in the case of Ethernet to token ring translation bridging, the bridge is also required to perform source routing.

b) *Spanning Tree Bridges :* In order to have a reliable network some fault tolerance is needed. This can be done by installing backup or secondary bridges in the network. But installing multiple bridges introduces the possibility of forming loops and hence non-delivery of frames. *Spanning tree bridges* are sophisticated bridges that communicate with each other using BPDUs (Bridge Protocol Data Units) and use the spanning tree algorithm to ascertain that no loops are formed. The spanning tree algorithm was given in section 3.1.1.1.

c) *Ethernet Address Blocking Or Filtering Bridges :* Some Ethernet bridges allow the network administrator to configure the bridge to drop certain Ethernet addresses. This is used to implement some level of security in the network or to limit access to improve network performance.

From now onwards, we will restrict our discussion to transparent bridges implemented in software and used to connect Ethernet segments.

3.3 Buffer Management Schemes

An important design aspect that influences the performance and behavior of bridges is the buffer management strategy. Ideally the buffer management scheme should be such that below a certain level of network traffic, no frames are discarded. And under heavy load conditions only as many frames are discarded so as to maintain network delay and traffic at some critical level. Further, frames should be discarded in accordance to some fairness criterion, such as, discarding frames that have not yet traveled very far. These objectives are, generally, difficult to achieve.

Existing literature [3, 13, 14] describes three basic buffer management schemes, depending on how the total number of buffers, are divided among the individual bridge output lines. Let b be the total number of buffers and l be the number of output lines. The three schemes are:

1. Complete Sharing : In this case the whole buffer space forms a single queue and is used on a first come first served basis.

2. Complete Partitioning : The b buffers are partitioned among the l lines, where each line has a dedicated access to b/l buffers.

3. Hybrid : The third case is when each line is dedicated b_i buffers. The various lines also share a common buffer pool.

The main strategies are summarized below, along with some of their advantages and disadvantages.

3.3.1 Complete Sharing

In this scheme, there is a single buffer queue which is used to store all incoming frames regardless of what port they are coming from. In general, it is the most efficient scheme in terms of buffer utilization. However, it has the drawback that it can result in a deadlock. A deadlock can occur when two or more nodes are unable to move frames due to unavailable space at all potential receivers. The simplest example of this is, two nodes *A* and *B* with frames for each other as shown in Figure 3-3. If all the buffers of *A* and *B* are full of frames destined for *B* and *A* respectively, then both these nodes are in a state of continuous deadlock. Because the transmitted frames do not find any space at the receiver. This situation can also occur in a more complex manner if two or more nodes forming a cycle are deadlocked. This occurs when their buffers are full of frames destined for other nodes in the cycle. This situation is depicted in Figure 3-4.

3.3.2 Complete Partitioning

As pointed out earlier, the simplest method of dedicating buffers to output lines

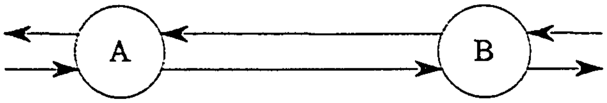


Figure 3-3 Simple Dead Lock

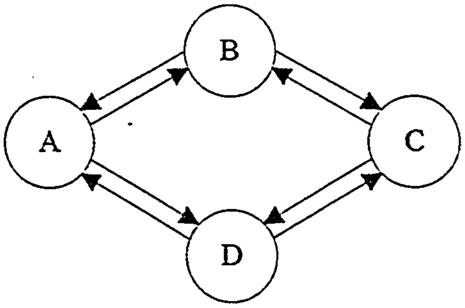


Figure 3-4 Complex Dead Lock

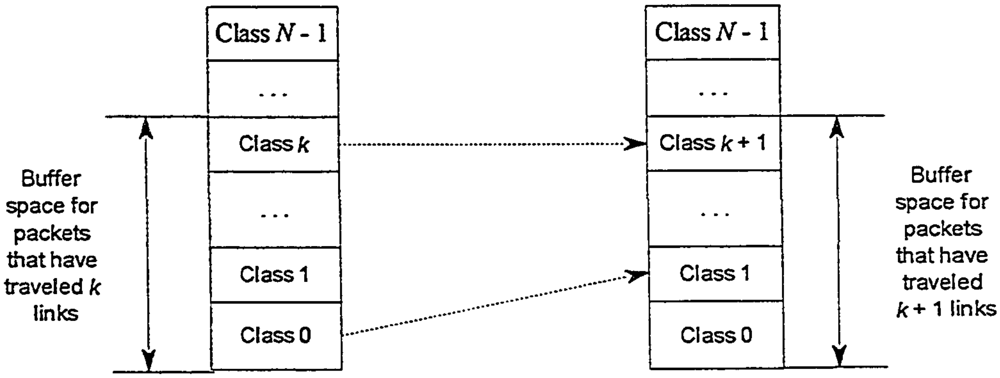

Class N - 1
. . .
Class k
. . .
Class 1
Class 0
Buffer space for packets that have traveled k links
Class N - 1
. . .
Class k + 1
. . .
Class 1
Class 0
Buffer space for packets that have traveled k + 1 links

Figure 3-5 Priority Queuing

is to divide them equally between them. No line may borrow a buffer from an idle line and is intuitively inefficient. A better allocation can be done if the traffic is divided into priority classes and space is allocated separately to each class as described below.

3 .3.2.1 Priority Queuing

For certain types of traffic, such as video and voice, an excessively delayed frame is useless, while in the case of a file transfer reliability is of greater value. Hence a notion of traffic classes is introduced. Another classification of traffic may be on the basis of the distance already traveled by the frame. Discarding a frame will probably cause the upper layers to retransmit it at a later time. Therefore discarding a frame that has already covered a large distance will only add to the stress on the network. A buffering scheme that takes various classes of traffic into consideration is a priority queuing scheme. Whatever the metric used, the buffer space is divided into different classes and the frames are stored in their respective buffer class (see Figure 3-5). Therefore, contention for buffer space occurs only within its class.

3 .3.3 Hybrid Queuing

In this scheme, a minimum number of buffers is reserved for each line with a limit on the maximum number of buffers that the line can attain. For each line, the optimal value of the queue length l is a complex function of the mean traffic. A simple rule of thumb has been determined which gives good but not optimal performance :

$l = b/\sqrt{s}$ [3]. For example, for seven pool buffers and three lines, $l = 7/\sqrt{3}$, i.e. four buffers are allocated.

3 .4 Packet Filtering

As more and more networks get connected together, the issue of network security becomes very important because the number of untrusted hosts and users connected to it increases. Security threats may come from within the organization or from outside. A security firewall is needed to protect from outside attacks. Packet filtering is one of the fundamental methods of protecting networks [15]. It consists of scanning the contents of the packet to determine, the packets source and destination addresses, its protocol type, and service. Then based on some security policy, the frame encapsulating the packet is either forwarded or discarded. Most of the screening information is contained in the different protocol headers of the packet.

First of all a security policy for the network is determined. In this it is decided what sets of addresses, protocols, and services are allowed, and what are not. In order to make sensible decisions to discard and forward packets, the internal workings of the protocols being filtered must be known. Not only the generic function, but even the behavior and peculiarity of the particular implementation used on the network need to be known. If the implementation behaves differently from the expected behavior then the packet filtering operation might not succeed in protecting the network, as otherwise expected. Some of these points are clarified in the example given in section 3.4.3. Packet filtering is a general procedure which can be used to make any network, using any protocol, secure. In this work, we confine ourselves to the objective of making a TCP/IP network secure from external threats i.e. creating an Internet firewall.

In order to determine a framework for protection, first the expected kinds of attacks need to be investigated. This is done in the next section. Then, different firewalling strategies are presented. Finally, the role of packet filters in this regard is discussed and illustrated with the help of an example.

3.4.1 Examples Of Network Attacks

Network attacks may occur because of flaws in particular implementations of software applications or because of inherent problems in the protocols themselves. A few protocol level problems concerning the TCP/IP suite of protocols, which are used throughout the entire Internet, are highlighted below for the purpose of illustration [15, 16, 17, 18].

1. A fascinating security hole was first described by Morris [18] in the TCP protocol to spoof a trusted host on a local network. The TCP connection is established via a three-way handshake. The client transmits an initial sequence number ISN_C. The

server acknowledges it and initializes its own sequence number ISN_S. Finally the client acknowledges the server's sequence number. The following is a schematic illustration:

TCP's three way handshake

Client ⇒ Server: SYN(ISN_C)

Server ⇒ Client: SYN(ISN_S), ACK(ISN_C)

Client ⇒ Server: ACK(ISN_S)

Client ⇒ Server: *data*

and/or

Server ⇒ Client: *data*

The Client must first receive the Server's ISN_S before any conversation can take place. This ISN_S is more or less a random number, but it is generated by an algorithm, thus it is possible to predict it. If an Intruder X could predict ISN_S, then it could send the following sequence to impersonate a trusted host, Target :

Intruder ⇒ Server: SYN(ISN_X), SRC=Target

Server ⇒ Target: SYN(ISN_S), ACK(ISN_X)

Intruder ⇒ Server: ACK(ISN_S), SRC=Target

Intruder ⇒ Server: ACK(ISN_S), SRC=Target, *nasty-data*

It is possible to predict the random number ISN_S because it is generated by incrementing the previous sequence number by a constant amount each second and by half that amount every time a connection is initiated. It is therefore possible to initiate a legitimate connection, observe the sequence number and then calculate the ISN_S for the next connection attempt.

The server's response will also be received by the target, which will try to reset the (illegal) connection with the server. Therefore, in response to the server's message to the target, the target itself tries to reset the connection. This was overcome by Morris by flooding the target with connection requests and thus generating queue overflows so that the server's response is not received by the target.

2. IP source routing mechanism can be easily abused [17]. The source is allowed to specify the return path for the IP packet. Using this mechanism it is possible that an intruder could specify a return path and use the IP address of a trusted machine. The source route specified would be different from the path normally taken to the targeted machine. All communication to the intruder will not be received by the target machine, and the intruder could easily impersonate the target machine.

3. The Routing Information Protocol (RIP) is a broadcast protocol that does not employ any validation. This protocol is used by gateways and hosts to construct routing tables to reach different destinations. An intruder can send bogus routing

information to a target host and each of the gateways along the path announcing a least cost path to another trusted host. Then the intruder could impersonate the trusted host. Another approach would be to route frames to the intruder for inspection or alteration and then send them back to the original intended recipient, using source routing.

3 .4.2 Firewalling Strategies

The literature describes three main strategies for firewalling : *packet filtering gateways*, *application level gateways* and *circuit level gateways* [15], each with its own advantages and disadvantages. The firewall may be implemented in critically located gateways or the hosts themselves.

Packet Filtering Gateways: Packet filtering gateways work on the basis of dropping packets based on their source or destination addresses, ports numbers, protocols and services. Generally no context is kept, and frames are discarded on the basis of decisions made from the contents of the current frame. Filtering may be done at the output time or the input time or both. The network administrator makes permission lists of acceptable hosts and/or services and stoplists for unacceptable hosts and services.

Application Level Gateways: Instead of using a general purpose mechanism, application level gateways use special purpose code for each application. The application may be an FTP session, a telnet session, an X windows session etc. Outbound traffic may be restricted to authorized individuals; all activity can be logged; effective bandwidth can be controlled to prevent bulk transfer of data [19]. The price paid is the need for a specialized user program or variant user interface for of the most services provided. This means that only the most important services will be supported. Further more, it will be harder to adopt newer technologies.

Circuit Level gateway: The third type of firewall gateway is the circuit level gateway. The caller connects to a TCP port on the gateway which, after validation, connects to some destination on the other side of the gateway. If the call is allowed, the gateway's relay program copies the bytes back and forth for that session. Generally, there is a verification protocol between the caller and the gateway. The protocol is used to describe the destination and service. If the session is allowed, the connection goes through, otherwise the gateway returns an error.

To allow applications to get through the gateway, modifications to the calling program or its library are required, which may be a problem. One way of doing this is to use replacement-system calls instead of the original calls *socket, connect, bind,* etc. (an example of a package providing such calls is *socks*, [15]). Source code availability and portability are issues of concern. In general, to make circuit level gateways more effective, some logging activity and connection controlling limits are also added.

3 .4.3 Example of a Packet Filtering Operation

As mentioned in section 3.4, in order to filter out packets of a particular protocol, the internal workings of that protocol need to be understood. Consider an example of a firewall that would allow telnet sessions to external machines but prevent telnet sessions to internal machines. Telnet uses the TCP protocol. To illustrate the difficulties of the process, first the communication model is briefly described. Then the relevant features of the TCP protocol are discussed. Finally the solution is presented.

Communication Model: This is a *client server* model. *Servers* are processes that provide some service via their TCP/UDP port numbers. The server processes are in a listening mode, waiting for a client's request. By convention, server port numbers are less than 1024. *Clients* use the services offered by the server, and are assumed to know the required port numbers of the server. Generally, the clients use whatever port number the local operating system assigns to them. A TCP circuit is identified by its four tuple :

(localhost, localport, remotehost, remoteport)

Where localhost and remotehost are the IP addresses of the local and remote machines respectively. Localport and remoteport are their port numbers.

TCP Protocol : The Transmission Control Protocol (TCP) provides a connection oriented service. It provides a reliable stream service by using mechanisms such as acknowledgments and retransmissions. The acknowledgment number gives the sequence number of the last successfully received byte. All packets except the very first TCP packet sent during a conversation contain an acknowledgment number. A sample TCP session is shown in Figure 3-6.

<u>Solution :</u> In a TCP conversation, packets are flowing in both directions. Even if all the data is flowing in one way, control and acknowledgment packets must flow the other way. Thus, uni-directional traffic cannot be used to establish a TCP session. Further, if the restriction is to allow packets flowing to port 23 (which is the telnet port number) of an external computer i.e. the circuits (Any-localhost, any-localport, any-remotehost, 23) is not adequate because there is no control on the external machine port 23. In other words, port 23 of the external machine could be used to establish subversive circuits to any of the internal machine ports. Fortunately there is a feature of the TCP protocol which can be used to establish the direction of the session. Only the first packet of the session contains no acknowledgment. All the rest of the packets contain it.

Hence the complete filtering process would be to scan octet numbers 12 and 13 (counting from 0) of the frame, to have the Hex value 0800. The scanning is done in network byte order. The value Hex 0800 signifies protocol type for an IP datagram. Then octet number 23 is scanned for the value 6, to determine that the packet conforms to the TCP protocol. After that octet numbers 36-37 (TCP dest. port, assuming no

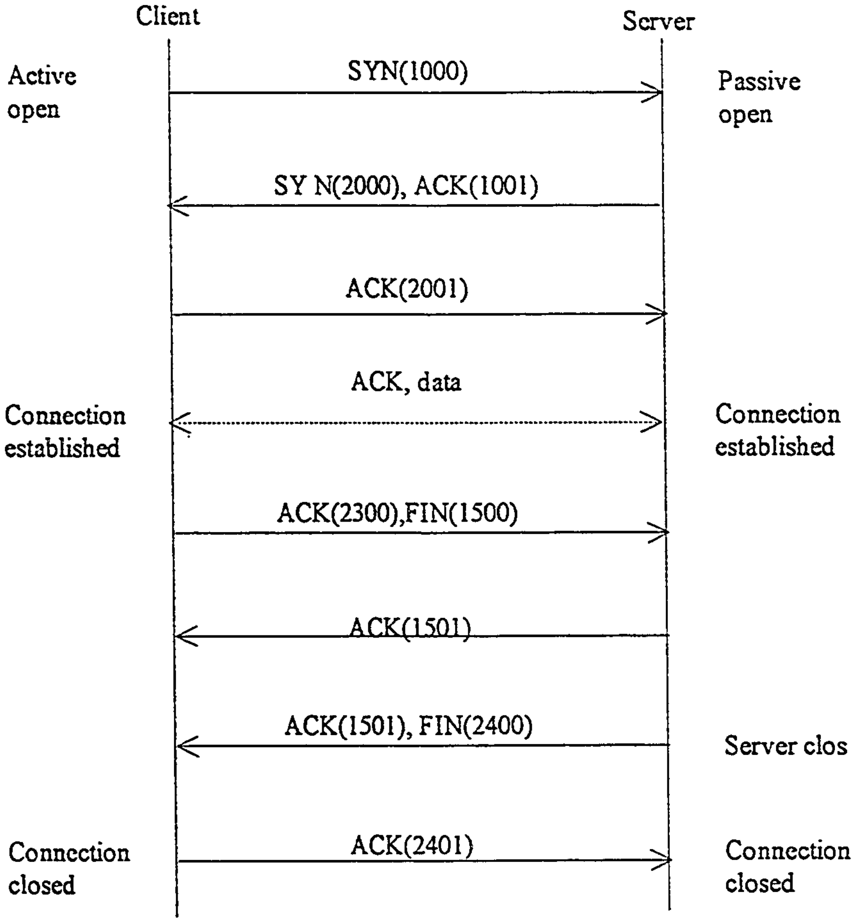


Figure 3-6 A sample TCP session

options are present) have the value 23 for telnet. Finally, octet numbers 42-45 are checked for the presence of an acknowledgment. If there is no acknowledgment and octet numbers 26-29 (IP source address) indicate an external source, the frame is dropped. All packets that have an external origin and contain acknowledgments are passed by the packet filter but are dropped by the server. The assumption is that the server would drop these packets because there is no proper TCP handshake.

3.5 Statistics Collection

Network traffic statistics have many important uses. Frame size and time distributions (which may have been filtered on protocol and service) are important when designing new protocols and network subsystems. Service usage statistics, per user, is used for accounting and billing. Protocol and service statistics is useful in planning network expansion and trouble-shooting.

Two types of statistics are of interest, one that conveys information about the electrical activity in the network and is generally retrieved directly from the interface. It includes information about erroneous frames, lost frames and cumulative network traffic in terms of frames and bytes. This information is required when planning for expansion or redundancy and also in trouble-shooting. The second type of statistics is generated by analyzing the data contents of the frame, which provides statistics on protocol and service usage. It can be generated by scanning specific octets of the frame as described in section 3.4.3. This type of statistics is useful in analyzing the protocol / service distribution of the network traffic, which in turn is useful in designing and expanding the network and also in the design and analysis of protocols.

Large corporate networks have multiple segments and use a number of protocols. To analyze the overall network traffic, statistics collection stations are put in each segment. They monitor the local traffic, perform some filtering operations and generate statistics according to some policy. The locally collected statistics are encapsulated in a datagram and sent to a central location to develop a global view (see Figure 3-7).

3 .6 Conclusions

In this chapter the IEEE recommendations for designing bridges were presented. Various types of bridges and their design issues were covered. The additional functions of packet filter firewall and statistics collection, which are incorporated in the proposed bridge were also presented. In the next chapter the actual design of the proposed bridge is presented.

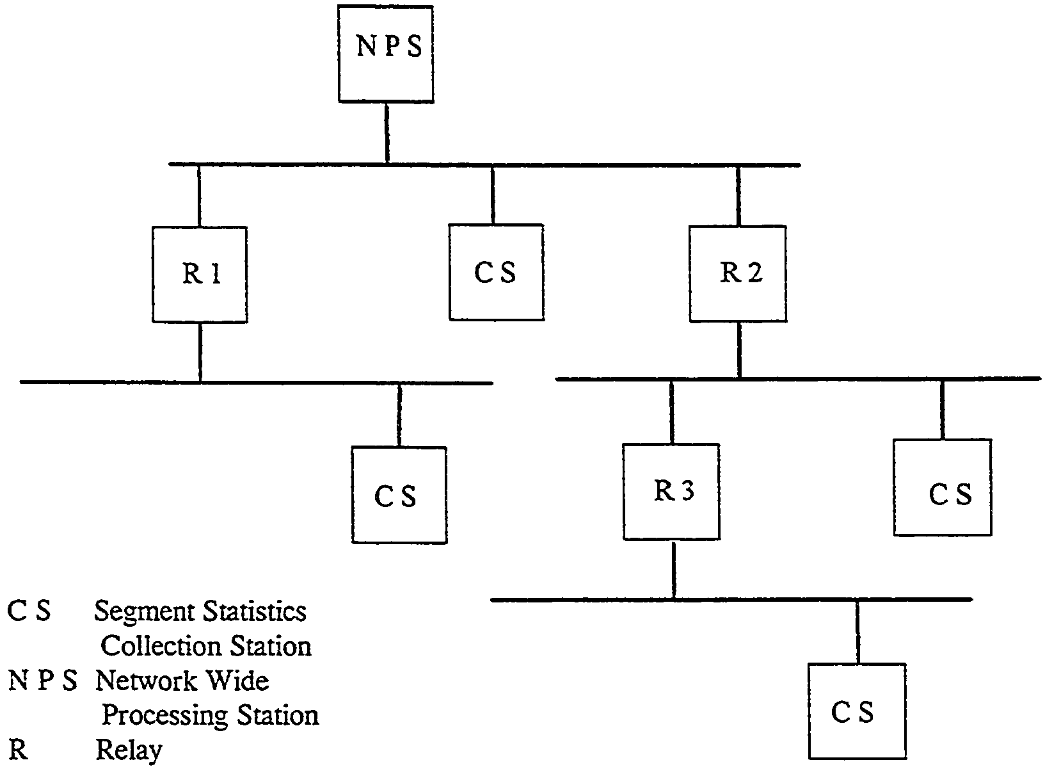

N P S
R 1
C S
R 2
C S
R 3
C S
C S
C S Segment Statistics Collection Station
N P S Network Wide Processing Station
R Relay

Figure 3-7 Network statistics collection

CHAPTER 4

4 DESIGN AND IMPLEMENTATION OF THE SOFTWARE BRIDGE

This chapter describes the design and implementation of the bridge and its packet filtering and statistics collection functions. It starts with the formulation of the complete design problem. Possible solutions are then identified along with the adopted solution and the reasons for its adoption. Finally it describes the implementation of the proposed bridge.

4.1 Design Overview

The design process can be broken down into the following four major steps:

- Selection of an interface specification standard and identification and implementation of required interface service primitives

- Bridge design
  - Buffer management
  - Learning algorithm
  - Database lookup / forwarding table
- Additional functionality
  - Packet filtering based on data content of frame (Firewall)
  - Statistics collection and analysis (network monitoring)

4.2 Selection Of A Specification Standard

Out of the three interface specification standards the Packet Driver specification was selected for implementation because of the following three reasons :

1 It is the only specification standard that passes the complete frame with hardware headers intact. This facilitates the use of a hashing function based on the Ethernet address bytes, for forwarding table look-up, as described later in this chapter.

2 Access to the complete frame makes the packet filtering task easier and more efficient.

3 It is a public domain specification standard (information is freely available).

The packet driver specification was studied and function calls were written according to the specification for use by the bridge and its additional functions.

4 .3 Proposed Bridge Design

In this section the major bridge design issues of buffer management and forwarding table lookup are considered. Various buffer management schemes have already been discussed in section 3.3. Some constraints on the queue design are imposed by the resources available for the implementation. This is discussed in the next sub-section. The number of buffers in the queue has been determined using the M/M/1 queuing model [14]. The parameters of the arrival and service processes were estimated based on extensive experimentation. Justification to use the M/M/1 queuing model and determination of the number of buffers is given in sub section 4.3.2.

Two choices are available for database lookup, (i) indexing and (ii) hashing. Any kind of indexing would be slower due to its indirect lookup nature. Further, the elements that we would like to lookup, that is, the Ethernet addresses, lend themselves to hashing. The hashing function should be efficient in terms of the number of table entries and ease of computation, and should not result in too many collisions. The hash function design is also discussed in this section.

4 .3.1 Buffer Management Scheme Used

A completely shared single queue, as described in section 3.3, is used. This is because the resources available for implementation, namely two NE2000 compatible network interface cards, cannot make use of the other strategies. The CPU is kept busy during most of the receiving interval by the receiver function. It is also kept busy during all of the transmitting interval by the sender function. This is explained below.

The NE2000 network interface card is Input /Output mapped (port mapped as opposed to memory mapped). On receiving a frame, the I/O mapped cards store it in their buffer and generate an interrupt. The CPU services the interrupt and copies the frame to the queue and updates some of the bridge variables. Hence, during this time it cannot send out frames on the other port even if that line is not busy. The packet driver specification allows a high performance function, *as_send_pkt,* which allows the packet driver and interface to take charge of the frame and free up the CPU even when the frame has not actually been sent. In other words the *as_send_pkt* function makes it possible to queue up frames for the interface, if the line is busy. The packet driver sends out the queued frames when the medium becomes available. Unfortunately this function is also not supported by I/O mapped cards. This means that, in our case, whether or not the transmission is successful the CPU is occupied and unavailable after it has attempted to send a frame. It remains tied up until the frame has been actually transmitted, or in case of an unsuccessful transmission, returned the error code.

The complete partitioning and hybrid buffering schemes are meaningful when transmission / reception of frames do not require the main CPU cycles. They are used to ensure that frames from a single line, destined to a busy line, do not overflow the total buffer space. Thus, making communication between some (other) idle lines impossible. Consider an example, for a two port bridge. A situation could be that for some time one of the segments, *A*, is busy and fills up the whole queue with frames destined to segment *B*. Further assume that when segment *A* becomes idle, *B* becomes busy trying to transmit to *A*. In the case of complete sharing, these frames are not forwarded to *A* because there is no buffer space to buffer these frames. But if there were some buffers dedicated to the port (either complete partitioning or hybrid buffering schemes), traffic from *B* to *A* could have been forwarded using these dedicated buffers. But as pointed out above, the resources used for implementation keep the CPU occupied during both the reception and transmission of frames. This means that (in the above scenario) while *B* is busy trying to send to *A*, the CPU can do nothing more than receive the frames and buffer them. So any advantage that may have been achieved by having dedicated buffers is lost. Therefore, the bridge uses a completely shared buffering scheme with a circular queue of length *b* buffers. Out of the *b* buffers *b - 1* are used for storing frames to be forwarded while one buffer is used for receiving the incoming frame which is later discarded. The reason for temporarily storing the frame is to be able to examine each incoming frame so as to update the interface's forwarding table and also collect the statistics from that frame.

4 .3.2 Size Requirement Of Buffers

Queuing Model: I/O mapped network interface cards were used in the proposed bridge. As they do not use DMA (direct memory access), queuing of frames to the port is also not possible (see section 4.3.1). Therefore, a single server representation is adequate. Secondly, it has already been discussed that the proposed bridge uses a completely shared single queue. Finally, it has been established in the literature [14] that the Ethernet traffic can be modeled by the Poisson process with exponential inter-arrival times. Hence the proposed bridge can be modeled using an M/M/1 queuing model.

Number Of Buffers: The number of buffers was determined by finding the actual arrival and service rates of the proposed bridge and plotting the utilization against the number of buffers, using the M/M/1 queuing model. The number of buffers for the M/M/1 queue is given by :

$$N = \frac{\lambda}{\mu - \lambda} = \frac{\rho}{1 - \rho}$$

where

$\rho = \frac{\lambda}{\mu}$ is the bridge utilization

μ = service rate in frames per second

λ = arrival rate in frames per second

Before proceeding further, some points are noted where the proposed bridge deviates from the theoretical model. Then the limitations of the resources used for determining the service rate, and how these limitations were overcome is described briefly. Finally, the service rate is determined and plotted against the utilization. This determines the minimum number of buffers that are required, to keep the *frame loss due to buffer scarcity* at or near zero.

Deviations from the theoretical model:

The service rate of the server should be independent of the arrival rate but that is not the case with the proposed bridge. When using the M/M/1 model, the complete bridge is represented by a single server and a single queue. The frames are assumed to get queued without the server's intervention. But in practice, when a frame arrives it generates an interrupt, which the CPU services. Hence the servers service is interrupted at each arrival. The number of interruptions depends on the size of the arriving frame and its arrival rate. These interruptions affect the bridge frame loss and forwarding delay characteristics. Ideally, the forwarding delay comprises of the transmission time and buffer residence time. In the proposed bridge the delay is a function of the transmission time, buffer residence time and the arrival process.

Limitations of resources used to determine service rate:

A traffic monitoring program, Netsight analyst, was used to generate and monitor traffic. Netsight analyst has the following limitations:

1. Due to copyright restrictions two copies of the program cannot be used on the same network.
2. The traffic it generates uses a fixed size frame, and during one session the generation rate is also fixed.

Due to Netsight's second limitation, fixed size frames and fixed rates were used to determine the maximum throughput of the proposed bridge at various frame sizes. This throughput was then mapped to a sample of actual network traffic to get a realistic distribution of frame sizes and thus an estimate of the average service rate. The service rate is defined as the maximum throughput in frames per second. The aggregate of two traffic samples was analyzed for fame size distribution. This is shown in Table 4-1. The service rate for various frame sizes was found, and is given in Table 4-2. Finally, the individual fixed frame size service rates were mapped to the sample of actual frame length distribution, see Table 4-3. The sum of the service rates of Table 4-3 is the service rate of the bridge. This is found to be 1512 frames per second.

The sample of actual traffic gives an average frame size 534 bytes. In Figure 4-1 the number of buffers is plotted against arrival rate (as a percentage of total Ethernet bandwidth), using the determined service rate and determined average frame size. The plot indicates that four buffers are enough for the proposed bridge up to a network load of 50%. In other words, if four buffers are used, frames will not be lost due to

Table 4-1 Sample of actual frame length distribution on college network

0-64 Bytes	64-128 Bytes	128-256 Bytes	256-512 Bytes	512-1024 Bytes	1024-1500 Bytes	Total frames	Avg. frame len.
1270	4120	180	136	502	2153	8361	534

Table 4-2 Service rates for different frame sizes at network load of 90%

Frame Size	64 Bytes	128 Bytes	256 Bytes	512 Bytes	1024 Bytes	1500 Bytes
Service rate at diff frm sizes (Max. Frms frwded)	1581	2097	1913	1278	742	513

Table 4-3 Mapped traffic

Frame Size	64 Byte	128 Byte	256 Byte	512 Byte	1024 Byte	1500 Byte	(μ) Total frms /sec
Number of frms frwded at max rates	2 40	1033	41	21	45	132	1512

Table 4-4 Arrival rate versus number of buffers

Normalized Arrival Rate	.10	.20	.30	.40	.50	.60	.61	.62	.63	.645
Number of buffers	0.18	0.45	0.87	1.6	3.4	13	23.8	35.4	41	696

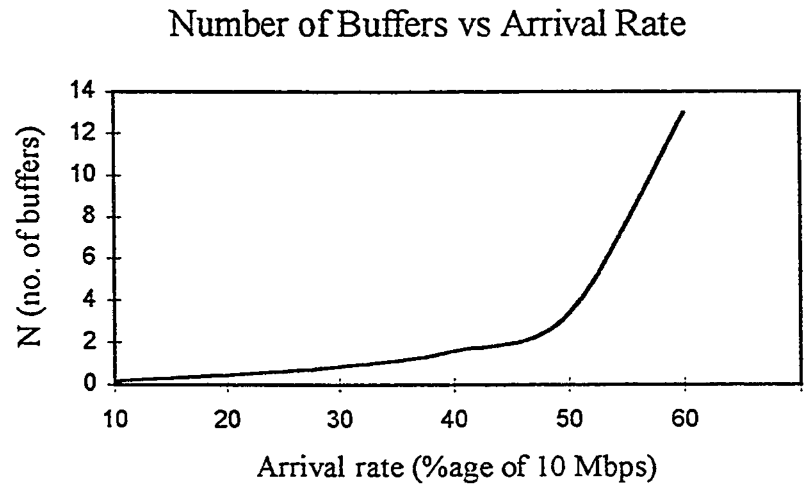


Figure 4-1 Frame arrival rate versus number of buffers

buffer unavailability up to a load of 50%. At 60% the requirement rises to thirteen buffers. Above 64.5% network load, the buffer requirement rises to infinity because the system becomes unstable, i.e. the arrival rate exceeds the service rate. Hence, about fifteen buffers is a good choice for the proposed bridge. This will enable the proposed bridge to take good care of occasional bursts of traffic of up to 60% network utilization.

4 .3.3 Hashing Function Used

Hashing function approach was adopted for efficiency reasons, (see section 4.2). Ethernet addresses have a total of forty eight bits. Part of the high order bits are vendor specific and the rest are the card's identification specific to that vendor. Three different hashing functions were tried for storing and retrieving Ethernet addresses in the address tables for the two ports. Conflicts were not resolved, instead table entries were overwritten with the new address value. The three hashing functions are:

1 Hashing on the least significant eight bits of the Ethernet address for a total of 256 address entries.

2 Hashing on the least significant sixteen bits of the Ethernet address for a total of 64K address entries.

3 Hashing on the summation of the Ethernet address bytes for a total of 1536 address entries.

The last two hashing functions performed reasonably well, i.e. not many frames that were not supposed to get across, were observed to get across the bridge once it had attained steady state (actually zero frames were observed). In the case of the first hashing function (hashing on the least significant eight bits) some frames did get across but still the number was less than one percent. This may be explained by the fact that the total number of machines and hence Ethernet addresses in the network used, is about a hundred and fifty. The population space is not large enough to justify definitive conclusions, but these results suggest that a summation of the Ethernet address bytes gives a reasonable hashing function. It neither has an excessive number of table entries nor does it have too many conflicts.

4.3.3.1 Bridge Forwarding

Two "source address tables", one for each port, are maintained by the bridging program. When a frame arrives, the source address table for the source port is checked to see if the destination address of the frame is in the table. If the address is there, the frame is discarded, otherwise it is forwarded onto the other port. Whenever a frame is received its source address is used to update the source address table entry. The proposed bridge also inserts a time stamp with the entry. The bridge's main program periodically checks all time stamps of the two tables and deletes the entries that have not been updated for about ten minutes (this is a tunable variable that can be set to a value of more than one hour). The value of the updating interval should not be too low because it will cause the tables to be flushed too often. This will make the proposed bridge forward frames that could have been discarded. On the other hand, an interval value which is too large will cause the bridge not to adjust quickly to computer relocation. This interval should be determined from the expected time of computer relocation. An estimate of this interval, based on experience, is in the range of ten minutes to half an hour. In order to relocate a computer from one room in a building to another, without reconfiguring the software, about ten minutes are required. If reconfiguration / re-installation of software is also required then the relocation takes more time.

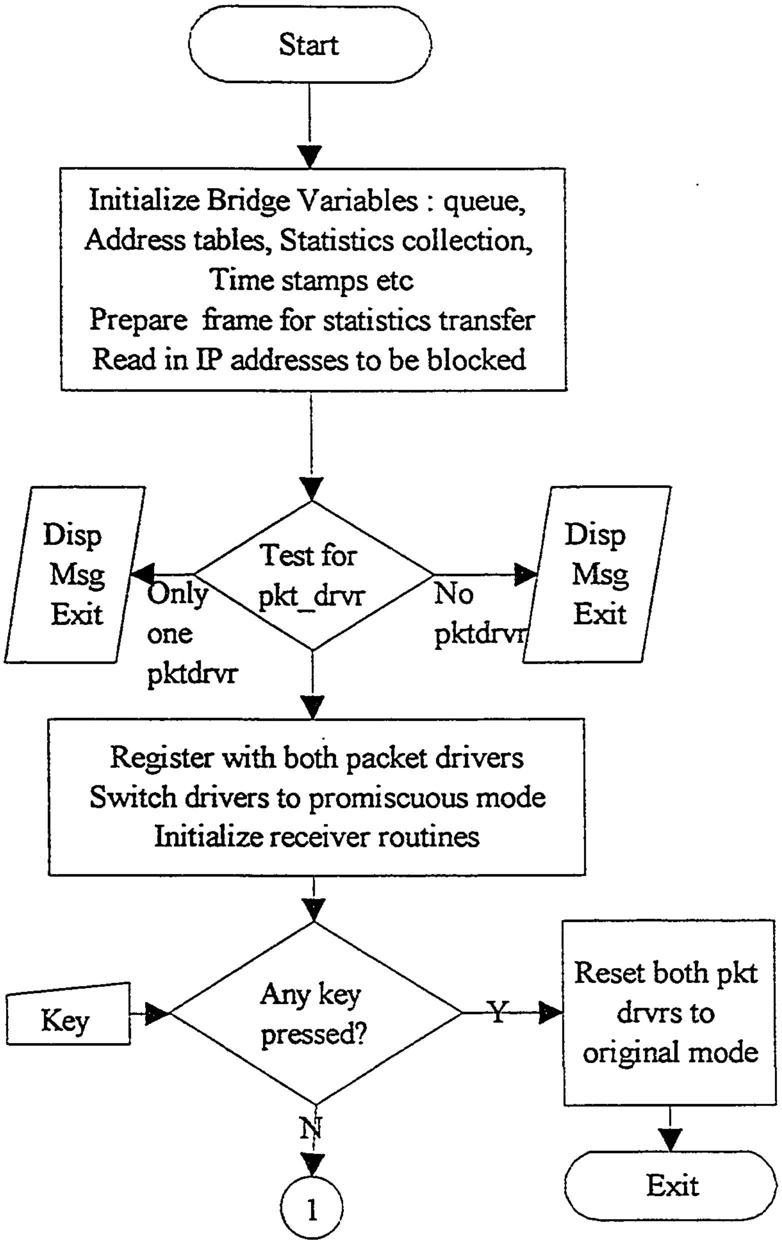
4 .3.4 Implementation

Bridge Initialization : As shown in the flow chart of Figure 4-2, the bridge first prepares the various data structures, that is, the queue to store frames, Ethernet address tables for each port, data structures for storing statistics, etc. Next, a dummy frame for sending out statistics is generated. This is done by reading in an IP datagram structure which is kept in a file (called stats_pkt). This structure is then modified to reflect the correct Ethernet address, IP address, length and IP checksum, UDP port number, etc., by the main program. The UDP checksum is calculated just before sending out each statistics packet, because its checksum includes the data field as well. In the book, *Networking with TCP/IP* [20], there is a detailed explanation on how to compute the UDP checksum. The statistics frame is broadcast, and is recognized by the signature “BRIDGE PACKET”, which occupies the first locations of UDP data area. The software then reads in the IP addresses to be blocked from a file *ip-stop*, which is located in the default directory (these addresses are to be firewalled).

Finally, the bridge software initializes both the cards into promiscuous mode (promiscuous mode receives all frames on the network regardless of destination). The software then registers appropriate interrupt handlers with the two interfaces, for receiving the frames, and goes into a infinite main loop.

Main Loop : After disabling further interrupts, the program sends out all frames accumulated in the queue. The first frame to be sent is at the bottom of the circular FIFO queue. It then checks to see if the time interval since the last statistics frame was sent, has exceeded the limit imposed by the sample time interval. If the answer is *Yes*, then it composes a statistics datagram from the dummy frame already prepared, fill it with required information, and transmits it. It then resets all statistics variables, starts the timer, and goes back to checking the queue. If the answer is *No*, then the program keeps on waiting for a frame to arrive.

Exit : If any key is pressed the program does a graceful exit by restoring the cards back to their original receive modes. Then it calls the *release_type()* function to unhook the receivers.


Start
Initialize Bridge Variables : queue, Address tables, Statistics collection, Time stamps etc
Prepare frame for statistics transfer
Read in IP addresses to be blocked
Disp Msg Exit
Only one pktdrvr
Test for pkt_drvr
No pktdrvr
Disp Msg Exit
Register with both packet drivers
Switch drivers to promiscuous mode
Initialize receiver routines
Key
Any key pressed?
Y
Reset both pkt drvrs to original mode
N
1
Exit

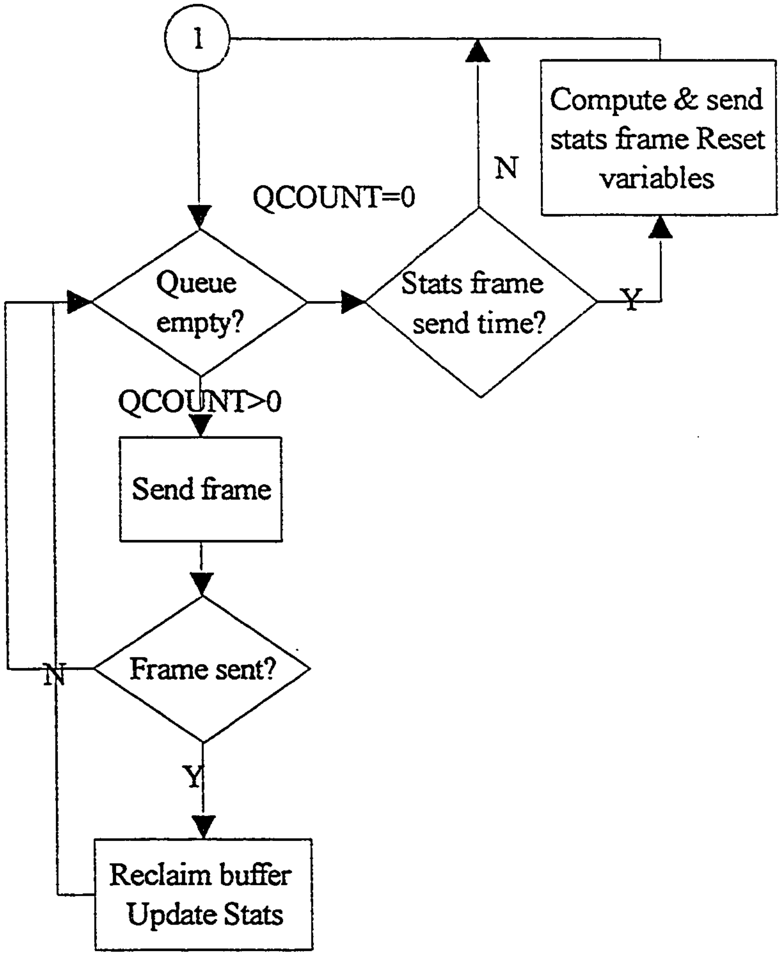


Figure 4-2 Bridge functionality flow chart

Interrupt Handler : When a frame arrives, the packet driver generates an interrupt which is handled by the interrupt handler for that card (see Figure 4-3). The handler passes on a buffer to the card to receive the frame. In the proposed scheme, a buffer is always available because out of a total of *b* buffers only *b - 1* may be used to store frames that are to be forwarded. One buffer is always kept free to be able to receive all frames, and at least update the interface tables and retrieve any statistical information.

The handler checks to see if the frame contains an IP packet (type=0x800) or not. If yes, then its destination IP address is compared with the blocked IP addresses. This is done in a sequential manner. If there is a match the frame is discarded. If the packet is not firewalled on the basis of its IP address then the handler checks to see if the frame should be forwarded or not, based on its Ethernet address. If the source and destination addresses lie in the same segment then the frame is discarded, otherwise it is stored for forwarding. Whether or not the frame is discarded, its Ethernet source address is used to update the interfaces table entry for future use. Before returning, the handler updates some variables for queue management and stores information to be used for statistics collection.

4 .4 Packet Filter

The bridge filters out frames carrying IP packets based on their destination IP addresses. The addresses to be filtered out are read from the file “ip-stop”, as

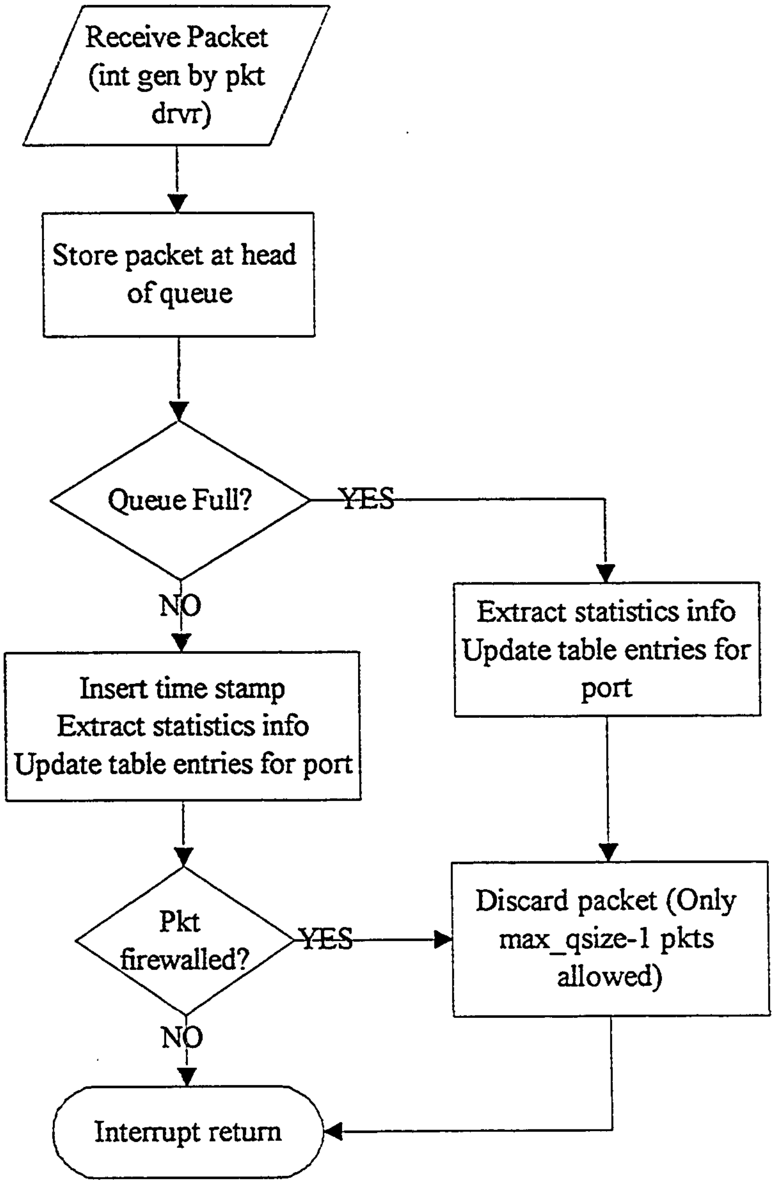

Receive Packet (int gen by pkt drvr)
Store packet at head of queue
Queue Full?
YES
NO
Extract statistics info Update table entries for port
Insert time stamp Extract statistics info Update table entries for port
Pkt firewalled?
YES
NO
Discard packet (Only max_qsize-1 pkts allowed)
Interrupt return

Figure 4-3 Flow Chart of Bridge's Interrupt Handler

mentioned earlier. To design the packet filtering function of the bridge, it needs to be decided what framing format to follow, either IEEE 802.3 with SNAP extension, Ethernet DIX, or both.

Ethernet DIX framing was adopted for the following reasons :

1. It is the most commonly used framing scheme for IP transmission on Ethernet networks.
2. It is very likely, that in any single location, there would be only one framing scheme used, either DIX or IEEE 802.3 with SNAP.
3. Due to the above two reasons checking for both framing schemes would only decrease the efficiency of the system.
4. The modification required in the software to filter IEEE 802.3 frames is not difficult. This is because it only means filtering on different octets of the frame.

The bridge checks each frame to see if it contains an IP packet. If the frame has an IP packet, it compares the packet destination IP address with the "stopped IP address list" in a sequential manner. If a match is found the frame is discarded, otherwise it is forwarded. To make the filtering operation efficient, a better algorithm such as hashing, would be required in place of sequential access. This task is left for future work.

4 .5 Statistics Collection Function Of The Bridge

Two types of statistics, as described in section 3.5 are collected by the proposed bridge. The physical statistics are taken from the network interface card and the statistics generated by filtering on the data contents of the frame, are collected by the bridging software. Further, the statistics are encapsulated in a UDP datagram and sent out periodically. The destination address and port numbers of the UDP datagram are configurable. The statistics frame is identified by the signature “BRIDGE PACKET” in the first thirteen data locations of the UDP datagram.

4 .5.1 Statistics Receiver

The statistics receiver and display program is much simpler than the Bridging program (see Figure 4-4 and Figure 4-5), but it uses essentially the same basic functions. The main differences are that it initializes the card to mode 3, which is used to receive only broadcast and frames destined to itself. It uses only two buffers, buffer 1 for holding a freshly arrived frame and buffer 2 to hold frames that have been confirmed to be carrying statistical information.

Once a frame arrives it is checked by the receiver to see if it has the statistics frame signature (BRIDGE PACKET) in the first data locations of the UDP datagram. If the answer is yes, it is copied to buffer 2 for display by the main loop, otherwise it is discarded. The main body of the program keeps displaying statistics from buffer 2, and does a graceful exit when any key is pressed.

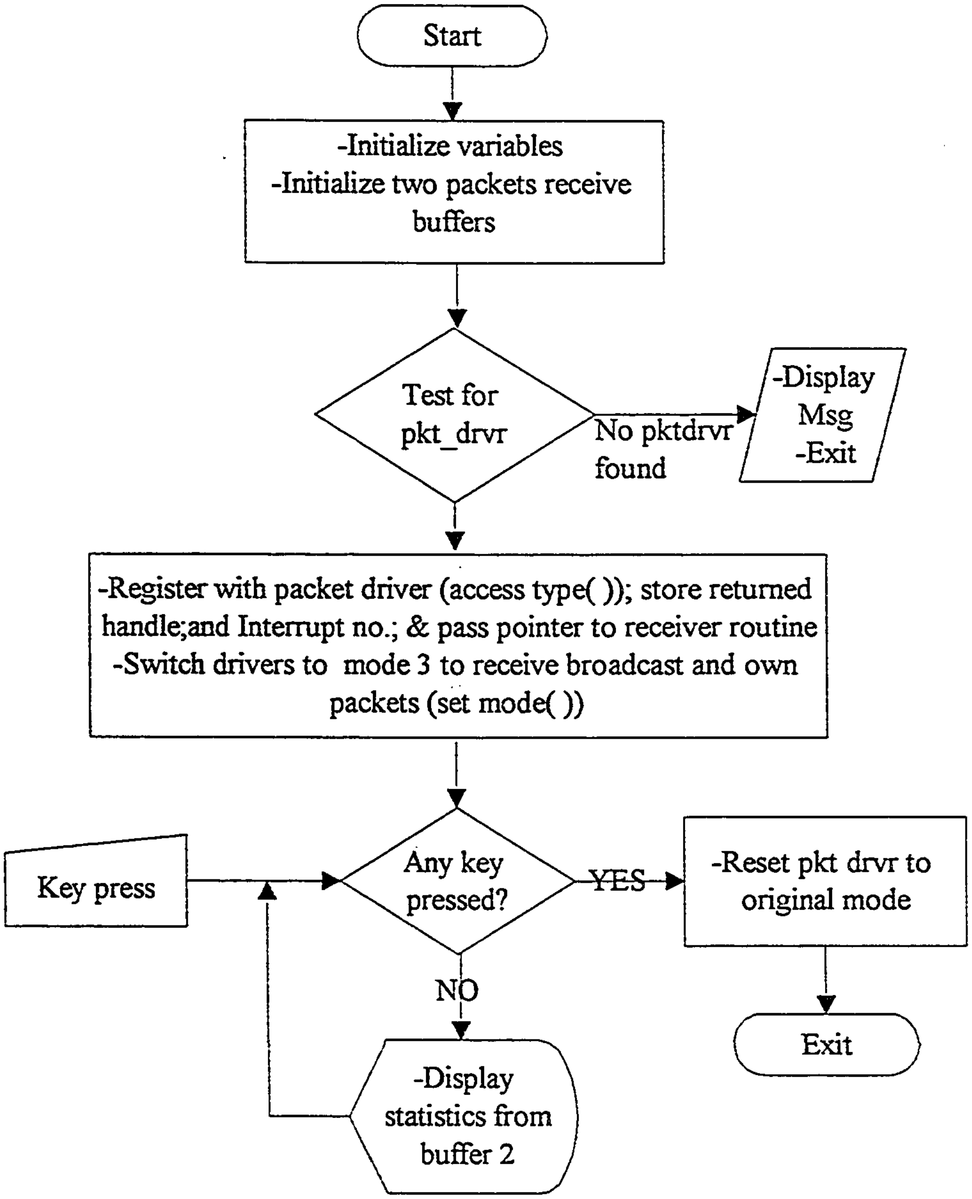

Start
-Initialize variables
-Initialize two packets receive buffers
Test for pkt_drvr
No pktdrvr found
-Display Msg
-Exit
-Register with packet driver (access type( )); store returned handle;and Interrupt no.; & pass pointer to receiver routine
-Switch drivers to mode 3 to receive broadcast and own packets (set mode( ))
Key press
Any key pressed?
YES
-Reset pkt drvr to original mode
Exit
NO
-Display statistics from buffer 2

Figure 4-4 Receiver for Statistics Frame

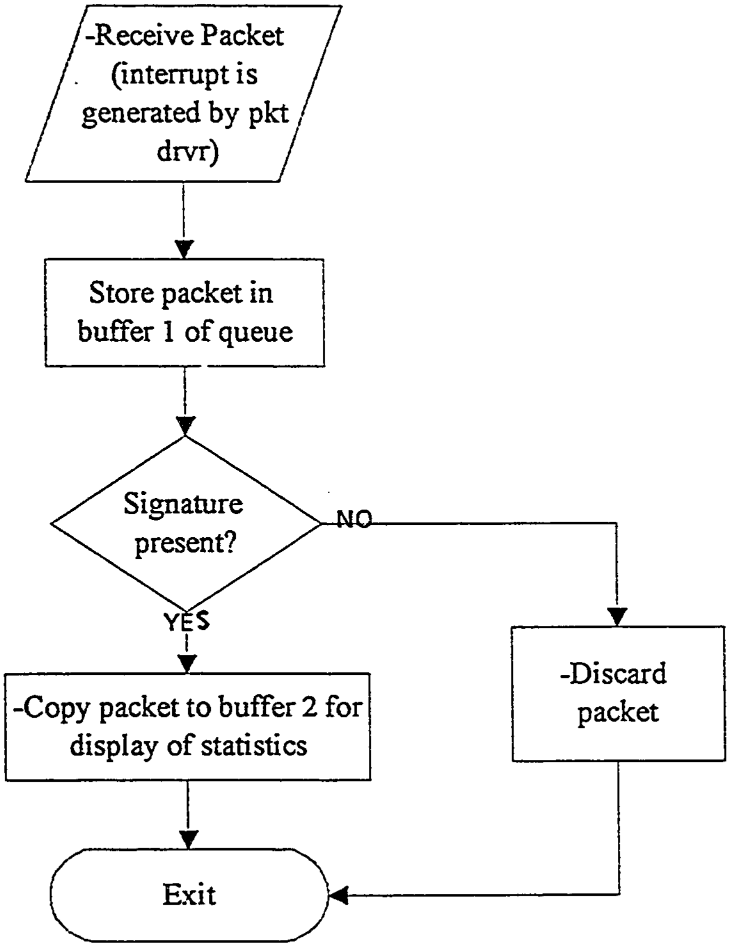

-Receive Packet (interrupt is generated by pkt drvr)
Store packet in buffer 1 of queue
Signature present?
NO
YES
-Copy packet to buffer 2 for display of statistics
-Discard packet
Exit

Figure 4-5 Receiver for statistics frame (interrupt handler)

CHAPTER 5

5 EXPERIMENTATION AND RESULTS

Performance tests have been carried out on the bridge and compared to the Ohio State University's KarlBridge program, and the College's Cabletron MMAC commercial bridge. Two types of tests were conducted. First, the forwarding abilities (frame loss) of each bridge was investigated and second, the forwarding delay imposed by each bridge was determined. Since no setup was available to measure the delay of the KarlBridge program, and the Cabletron MMAC, the delay performance test was done only on the proposed bridge.

The relationship between the number of buffers and the frame loss, as well as, the number of buffers and delay of the proposed bridge were separately investigated.

Before proceeding with the test results, the experimental setup is described in the next section followed by a comparison of the architectures and capabilities of the bridges used in the tests. Finally, the test results and their analysis are presented.

5.1 Experimental Setup

To test the forwarding abilities, the bridges were placed between a station generating controlled traffic at various frame rates and sizes and a network monitoring station at the other end. Both traffic generation and monitoring was done using Netsight Analyst. The complete session time was calculated from the difference of time stamps of the first and last frames. The total number of frames was divided by the time difference to obtain the output rate of the proposed bridge. Some important points regarding the test conditions are given below:

1) The traffic generated is not typical Ethernet traffic, because:

a) There are no collisions, a single station is generating frames at a specific constant rate.

b) No traffic is entering the bridge from the monitoring end, which means that while receiving a frame another one does not arrive on the other port and initiate its own interrupt.

2) One important difference in testing the Cabletron MMAC and the software bridges was that the two software bridges were completely isolated when tested, while the Cabletron bridge was still connected to the network but at a very light load.

3) In order to make a fair comparison, the filtering capability of KarlBridge was turned off during the test.

4) The "forwarding delay" is the delay in microseconds from the time the first bit of a frame is received on one port to the time the first bit of the same frame is transmitted on the other port. This delay varies depending upon the bridge architecture and algorithm. An ideal bridge is one in which the instant the frame is received, it is transmitted out on the other port. Thus the minimum frame forwarding delay will be larger for larger frames because they take longer to receive.

The forwarding delay of the bridge was measured by inserting time stamps on receipt of each frame and computing the difference when that frame was transmitted. This is done by the bridging software itself. Two points, however, need to be noted: first, the time stamp is inserted after the receipt of the frame and compared to the time after the frame has been transmitted completely. This does not affect the calculations of the forwarding delay because injection rates are equal at both ends. Secondly, the time stamp is inserted *after* the filtering routines and therefore ignores the filtering routine execution time. However, the filtering routine execution times for any subsequent frames that arrive, while this frame is in the queue, are included in the calculation of the forwarding delay.

5) The time resolution provided by the computer timer is not high. It is eighteen ticks per second. This means that in one tick, more than forty six maximum sized frames can be transmitted at 100% network utilization. The effect of low timer resolution is that delay values are not very precise. To minimize the error introduced by this factor, large samples were taken (approx. ten thousand frames) and the samples were averaged over three readings.

5 .2 Characteristic features Of the Bridges Tested

Table 5-1 lists various features of the three bridges which were tested and compared. The Cabletron MMAC is a six port commercial hardware bridge while the proposed bridge and KarlBridge are software bridges. The proposed bridge and KarlBridge compare well in their features. Both are two port bridges, do some level of packet filtering, and use PC hardware on the DOS platform. The KarlBridge provides some SNMP support while the proposed bridge conforms to interface specification standards.

5 .3 Bridge Performances Results

Three types of tests were conducted on the proposed bridge. First, its

Table 5-1 Comparison of bridges tested

	Proposed Bridge	OSU's KarlBridge	Cabletron MMAC
Implementation	Software	Software	Hardware
Ports	2	2	6
Packet Filtering	yes	yes	no
Follows interface spec. std.	yes	no	NA
Dedicated Hardware	no	yes	yes
SNMP Support	no	yes	yes
Drop Pkts. Imdt. if Filtered	configurable	yes	NA
Spanning Tree / Source Routing	no	no	no
Hash Table Size	1.5k / 64k	4k	NA

forwarding abilities were tested and compared with those of the other two bridges. Second, its forwarding delay was determined and compared. Third, the proposed bridge was connected in the network and its overall performance analyzed.

5.3.1 Forwarding Capabilities

The bridges were tested at three different network loads and at several frame sizes. Table 5-2 shows the frame forwarding rates of the bridges at a network load of twenty percent, Table 5-3 shows the bridge performance at forty percent. These two tables represent the typical range of network operational load, since above forty percent utilization, performance starts to decrease as the utilization increases (because of increased collisions for the CSMA/CD protocol). For this reason, Ethernet networks are rarely used where the average network utilization is above 40%. As can be seen from the tables, the proposed bridge performs well in forwarding frames of all sizes at acceptable Ethernet loads.

For frame sizes greater than 256 bytes and Ethernet load of forty percent, all bridges forward all frames arriving at their ports, i.e., no frames are lost. The performance of all three bridges falls, with nearly equal magnitude, at small frame sizes of 256 bytes or less, to about 44% for a frame size of 128 bytes and 72% at 64 bytes. This is expected behavior, because with smaller frame sizes the number of interrupts to the system and hence the processing overhead increases for the same number of bits forwarded. On the other hand, this does not severely impair the performance of the bridge since most protocols put a limit on the minimum size of the frame because it is inefficient (high overhead) to use small frames. The information given in Table 5-2 and Table 5-3 is plotted in Figure 5-1 and Figure 5-2. Notice that the curves of our proposed bridge and KarlBridge overlap.

Table 5-4 and Figure 5-3 show frame loss percentage, for various frame sizes at 90% network load. The performance was tested using a single source and a single destination with the bridge in between, hence, it reflects the bridge algorithm performance. Since our proposed bridge uses an additional layer of abstraction and conforms to the packet driver specification, it was expected to be somewhat inferior in performance. As can be seen from Figure 5-3, the proposed implementation is very slightly inferior at frame sizes of 256 to 1024 bytes. But the performance is well within acceptable limits. As discussed later under "Suggested Possible Applications" in the next chapter, this price is well worth paying. While viewing the results, the following differences need to be kept in mind: the proposed bridge is a two port while the Cabletron MMAC is a six port bridge. Also, in the experimental setup the two software bridges were completely isolated when tested, but the Cabletron bridge was still connected to the network but at a very light load.

Table 5-2 Frames forwarded per sec for several frame sizes at network load of 20%

Frame Size	1500 Bytes	1024 Bytes	512 Bytes	256 Bytes	128 Bytes	64 Bytes
Frms injected / sec	167	244	488	977	1953	3906
<u>Frms forwarded / sec</u> Our Bridge	167	244	488	977	1953	2479
Cabletron's MMAC	167	244	488	977	1952	2182
KarlBridge (OSU)	167	244	488	977	1952	2329

Table 5-3 Frames forwarded per second for several frame sizes at network load of 40%

Frame Size	1500 Bytes	1024 Bytes	512 Bytes	256 Bytes	128 Bytes	64 Bytes
Frms injected / sec	333	488	976	1953	3906	7812
<u>Frms forwarded / sec</u> Our Bridge	333	488	976	1913	2172	2217
Cabletron's MMAC	333	488	976	1806	2023	2216
KarlBridge (OSU)	333	488	976	1917	2156	2216

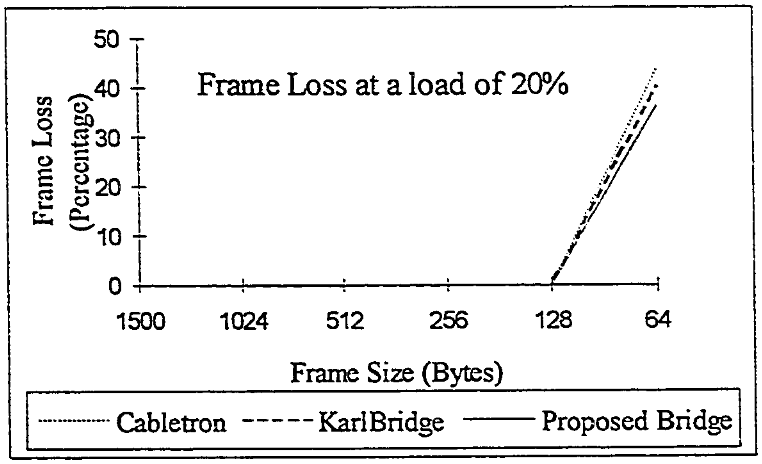


Figure 5-1 Frame loss as a percentage, for various frame sizes at 20% network load

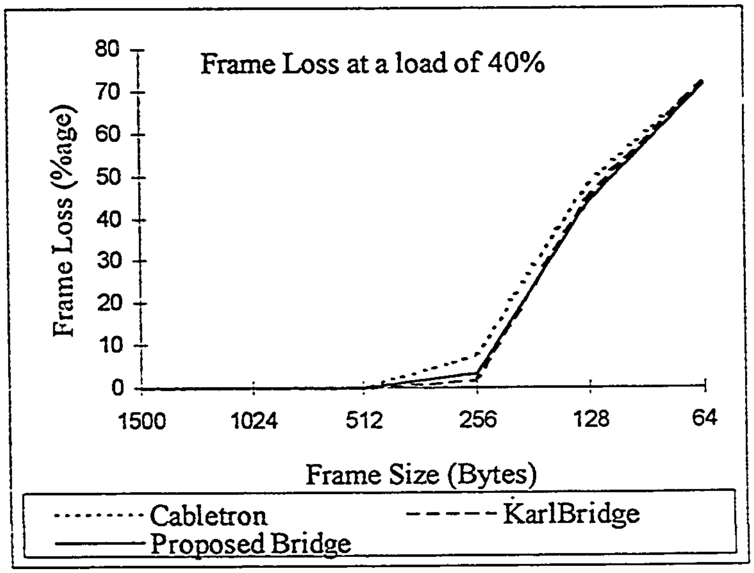


Figure 5-2 Frame loss as a percentage, for various frame sizes at a network load of 40%

5 .3.2 Delay Comparison

The graph of Figure 5-4 shows the delay that the proposed bridge imposes on a given frame that is forwarded through it. It was difficult to measure the delays of the Cabletron and KarlBridge due to lack of resources. The source code of the PCbridge program was not available. Thus the delay testing was done only on the proposed bridge.

The delays for KarlBridge and PCbridge of NorthWestern University have been reproduced from reference [2]. As can be seen from the graph of Figure 5-5, the proposed bridge has good delay performance when compared with other software implementations.

During tests it was observed that as the frame sizes got shorter there was a rise in the delay followed by a decline. In other words, keeping the input bit rate constant, as the frame size is decreased, there is a fall in the forwarding delay up to a certain point followed by a rise and then a fall again. This phenomenon can be explained as follows. The expected behavior is that the delay should keep on decreasing with a decrease in frame size, because the bit transmission time is smaller for smaller frame sizes. Ideally, delay is comprised of the transmission time of the frame and the queue residence time of the frame. However, in the proposed design, another factor, namely

Table 5-4 Frames forwarded per second for several frame sizes at a network load of 90%

Frame Size	1500 Bytes	1024 Bytes	512 Bytes	256 Bytes	128 Bytes	64 Bytes
Frms injected / sec	750	1098	2197	4394	8789	17578
Frms forwarded / sec						
Our Bridge	513	742	1278	1913	2097	1581
Cabletron's MMAC	517	747	1418	1780	2099	1582
KarlBridge (OSU)	517	747	1431	1890	2099	1581

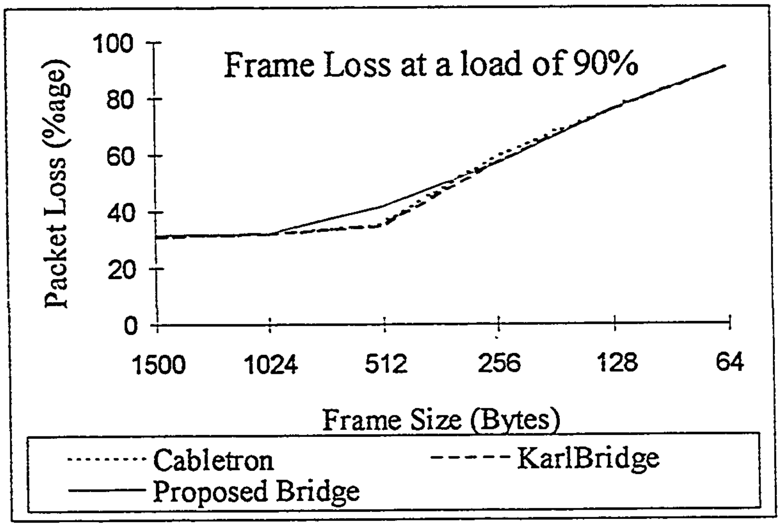


Figure 5-3 Frame loss as a percentage, for various frame sizes at 90% network load

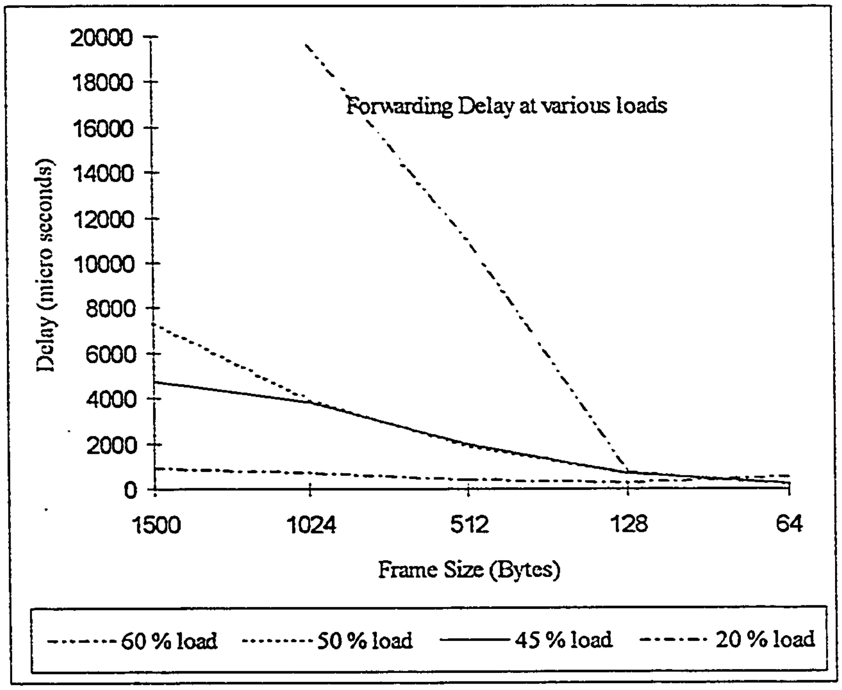


Figure 5-4 Forwarding delay of proposed bridge at various loads

the number of interrupts to the system, also contributes to the delay. Whenever a frame arrives, an interrupt is generated which is serviced by the CPU. Thus the service rate is not independent of the arrival rate. Therefore, shorter frame lengths result in higher arrival rate, causing a larger number of interrupts.

The delay performance was thoroughly investigated. It was found that the "hump" in the delay characteristics is not stationary and is observed at the point where the system starts to loose frames, during its forwarding process. This point depends on the arrival rate and the frame size. Figure 5-6 shows the delay characteristics for a network load of ten, sixty, and seventy percent, respectively. Figure 5-7 shows the forwarding characteristics for the same load. It is observed that at low load (≤ 10%) the delay plot has no hump. This is because there is no forwarding loss at ten percent or lower loads. For sixty and seventy percent of network load there is a frame loss from the very beginning. Hence the point where the proposed bridge moves from the "no loss state" to the "lossy state" is not seen and the delay curve has no hump too. Figure 5-8 through Figure 5-12 show the delay characteristics for network loads of twenty, thirty, forty, forty five and fifty percent, at various frame sizes. Figure 5-13 shows the forwarding characteristics for these loads. It is seen from these figures that the "hump" in delay curves occurs at the point where the system changes from "no loss" state to "loss" state. Further, recall from Figure 4-1 that the proposed bridge requires less than three buffers to forward frames without incurring any losses for a network load of up

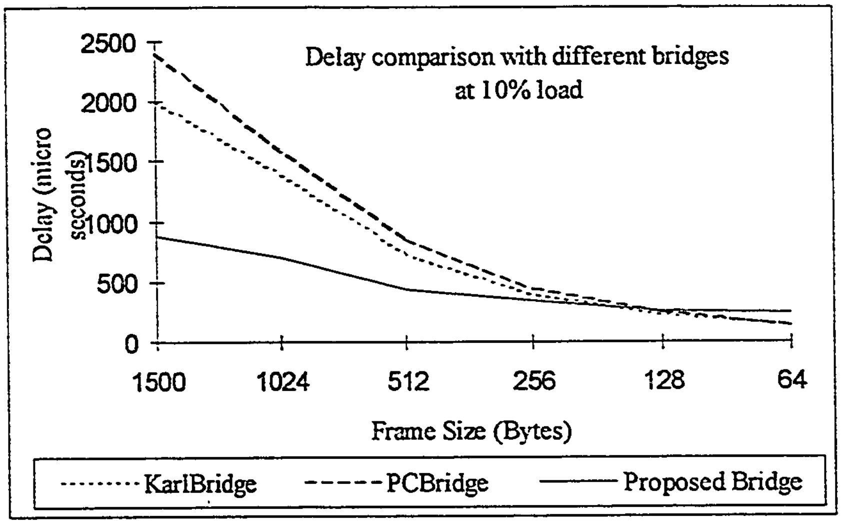


Figure 5-5 Forwarding delay of the bridge compared to KarlBridge and PCbridge at 10% network load

Table 5-5 Forwarding delay at various frame sizes and different network loads

Frame Size (bytes)	1500	1024	512	384	256	192	128	96	64	32
<u>Delay (in micro sec) at :</u> 10 % load	880	700	440	X	350	X	260	X	245	X
20 % load	910	710	430	X	330	X	307	685	565	530
30 % load	880	610	460	X	535	950	5420	710	440	X
40 % load	1000	850	850	880	6830	6570	720	X	240	X
45 % load	4750	3830	1970	1550	7850	6670	700	X	250	X
50 % load	7330	3950	1920	8920	7670	6750	730	X	240	X
60 % load	27820	19660	11020	X	7640	X	750	X	230	X
70 % load	28350	19940	10990	X	7500	X	750	X	250	X

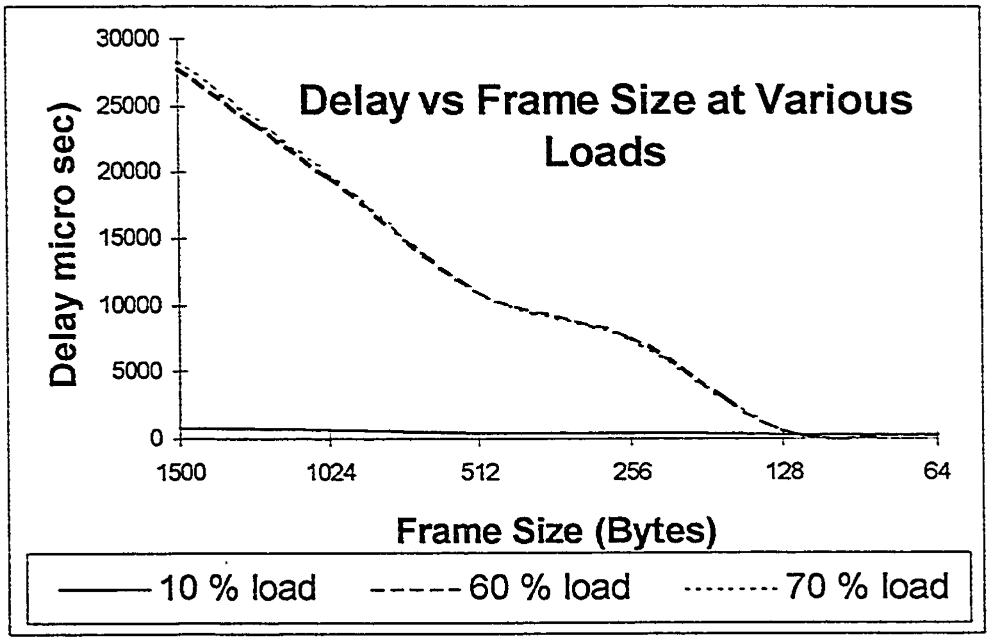


Figure 5-6 Forwarding delay at 10, 60 and 70% load

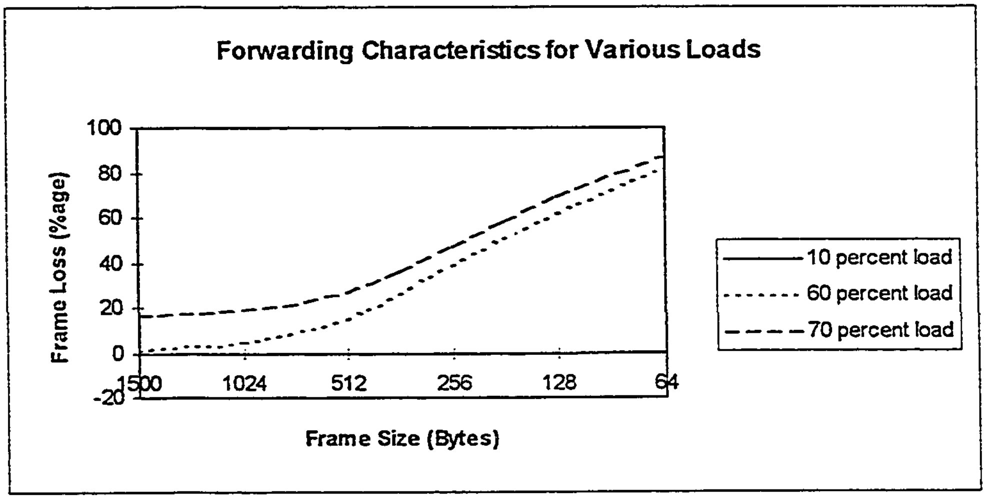


Figure 5-7 Forwarding characteristics at 10, 60 and 70% network loads

Table 5-6 Frame loss as a percentage at 10, 60 and 70% network loads

Frame Size	1500	1024	512	256	128	64
10 percent load	0	0	0	0	0	0.3
60 percent load	0.9	4.6	15	39	62	82
70 percent load	17	19	27	47	69	87

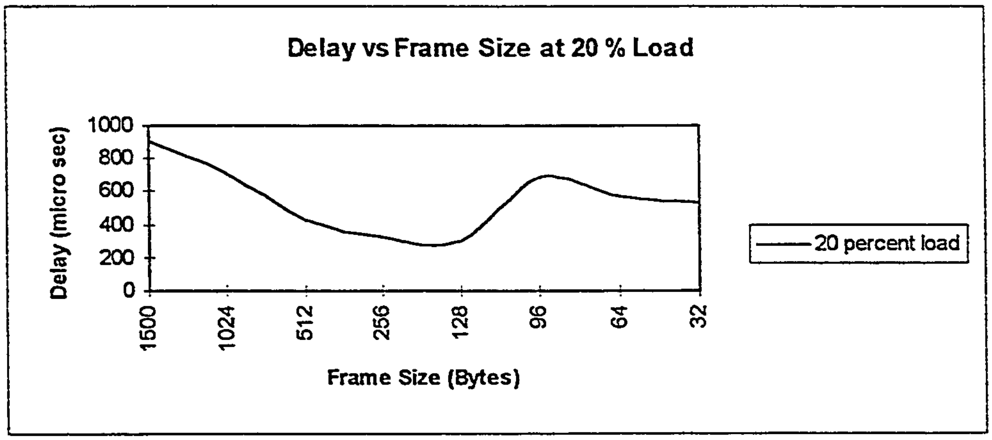


Figure 5-8 Forwarding delay at 20% load

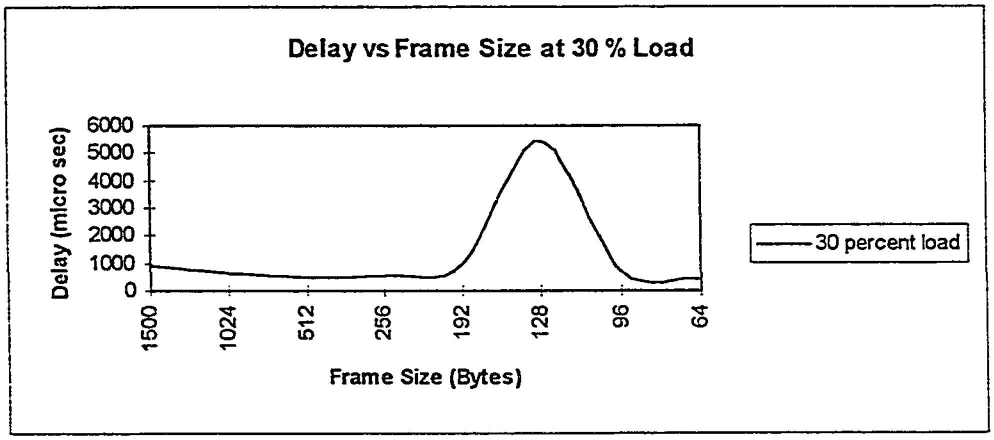


Figure 5-9 Forwarding delay at 30% load

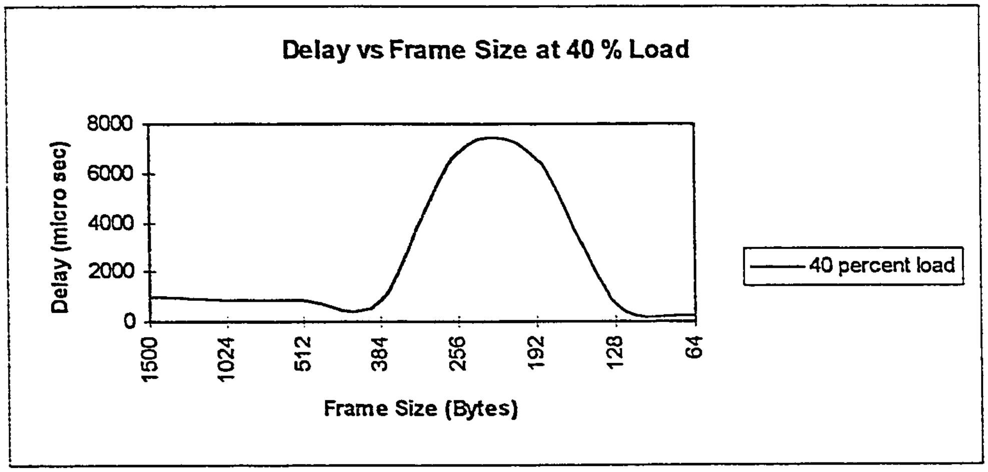


Figure 5-10 Forwarding delay at 40% network load

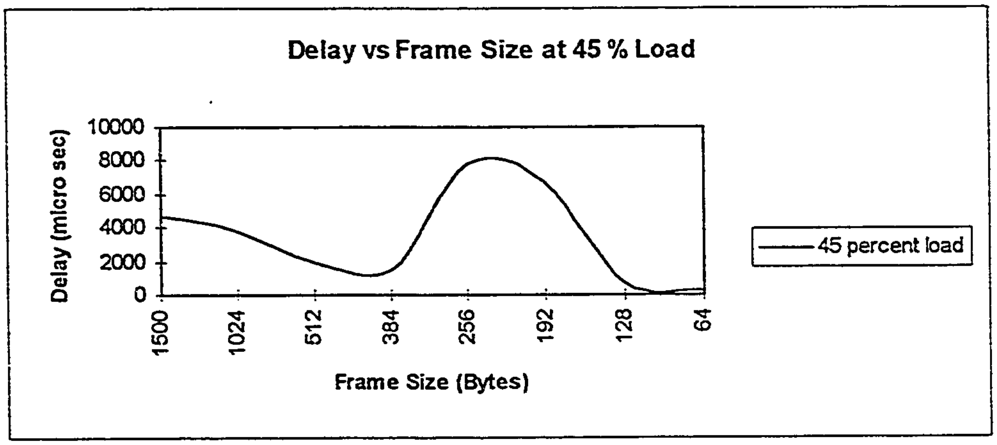


Figure 5-11 Forwarding delay at 45% network load

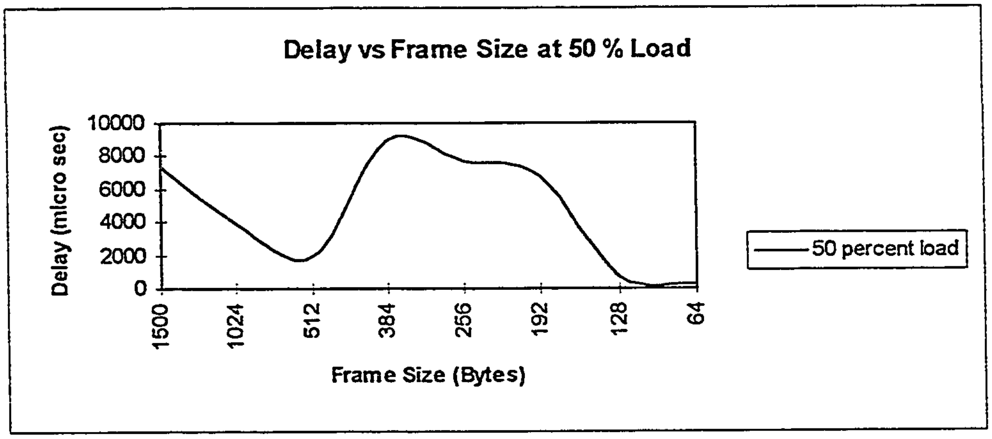


Figure 5-12 Forwarding delay at 50% network load

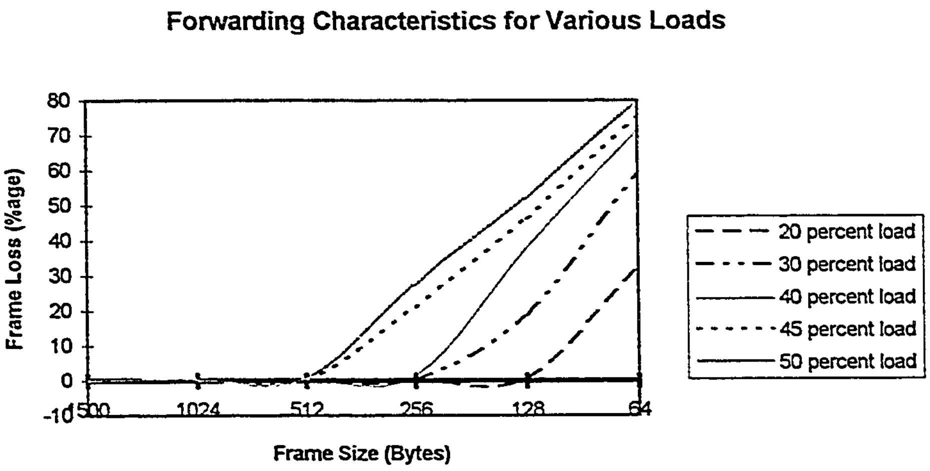


Figure 5-13 Forwarding characteristics at 20, 30, 40, 45 and 50% network loads

Table 5-7 Frame loss as a percentage at 20, 30, 40, 45, and 50% network loads

Frame Size	1500	1024	512	256	128	64
20 percent load	0	0	0	0	0.3	32
30 percent load	0	0	0	0.25	18.4	58.7
40 percent load	0	0	0	1.5	38	71.7
45 percent load	0	0	0	21	46	75
50 percent load	0	0	0.35	27.4	52	79.3

to fifty percent. At sixty percent load this requirement jumps to thirteen buffers and is six hundred and ninety six buffers at sixty four percent network load. In other words, the buffer requirements of the proposed bridge are less than three buffers for most of the working range and rise very sharply, just before the system goes into the unstable state, where arrival rate exceeds the service rate. The experiment has been carried out with the proposed bridge having fifteen buffers. It is concluded from the above, that just before the system enters the "lossy" state the buffer requirements and hence the buffer occupancy distribution of the system changes sharply, meaning that more frames get piled in the queue. This gives rise to the "hump" in the delay characteristics at this same point. As mentioned earlier, the delay is also a function of the number of interrupts to the system (and therefore the arrival rate) and the processing routine execution times.

Figure 5-6 shows that for high loads (60 and 70 %) the delay characteristics remain nearly identical. This is explained as follows. A closer look at the situation reveals that the frame sizes and hence the transmission times are identical. The processing routine execution times are constant and independent of the size of the frame or its arrival rate. However, the number of interrupts to the system is different. This is because at seventy percent load the number of frames per second is larger than that at sixty percent load..But because the interrupts are disabled during both, the arrival and sending routine executions, and also due to the fact that a maximum of only one interrupt is queued. All extra interrupts due to the difference between 60 and 70%

loads get discarded. This is true in all cases where the system is in a loss state through out the test. In this situation the system is forwarding at it's maximum service rate for the loads and hence, the frame output rate is identical.

5 .3.3 Effect Of Number Of Buffers On Bridge Performance

As can be seen in Figure 5-14, and Figure 5-15 as the number of buffers increases so does the forwarding delay. This is in accordance with Little's law. In Figure 5-16 it can be seen that increasing the number of buffers has no effect on the bridge frame loss performance, except at very low buffer number. This is in complete agreement with the design decision for the number of buffers, as discussed in section 4.3.2.

5 .3.4 Other Experiments

The bridge was connected so that all the traffic coming to the college lab servers passed through it. It performed correctly and reasonably well in its basic bridging function. No "unwanted" frames were observed after the bridge had acquired steady state when the hashing was on the low order sixteen bits of Ethernet address or on the summation of Ethernet address bytes. When hashing was on the low order eight bits some "unwanted" frames were observed to get past the bridge.

Table 5-8 Delay versus the number of buffers

No. of Buffers	2	5	10	15	20	25	28
Delay at 40% load (μ sec)	710	710	770	830	840	870	880
Delay at 60% load (μ sec)	850	5125	12390	19680	26950	34225	38550
Delay at 80% load (μ sec)	850	5200	12590	19960	27300	34700	39100

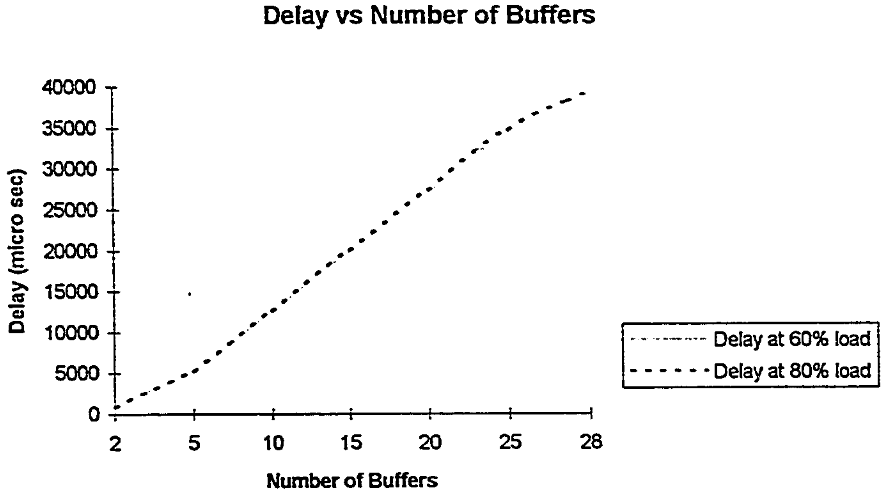


Figure 5-14 Delay versus number of buffers for 60 and 80% utilization

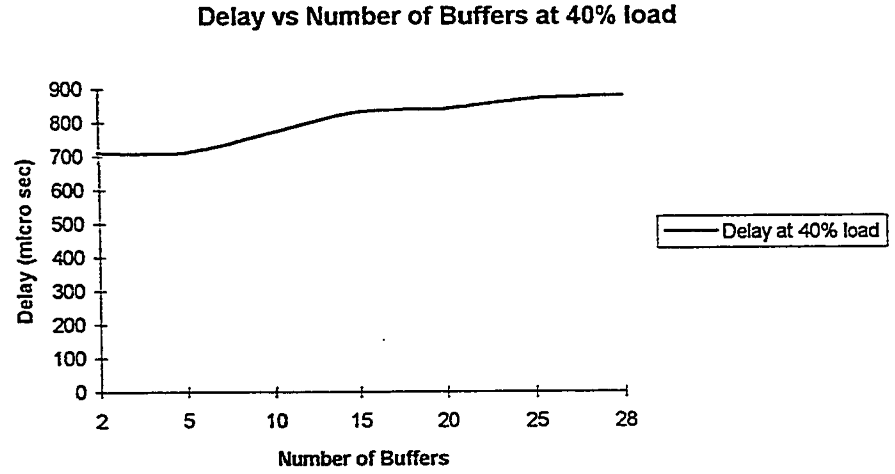


Figure 5-15 Delay versus number of buffers for 40% utilization

Table 5-9 Frame loss versus number of buffers

No. of Buffers	2	5	10	15	20	25	28
Frm loss at 40% Utilization	0.16	0.1	0.045	0	0	0	0
Frm loss at 60% Utilization	5.5	4.78	4.6	4.6	4.6	4.6	4.6
Frm loss at 80% Utilization	30.1	29.4	29.4	29.4	29.4	29.4	29.4

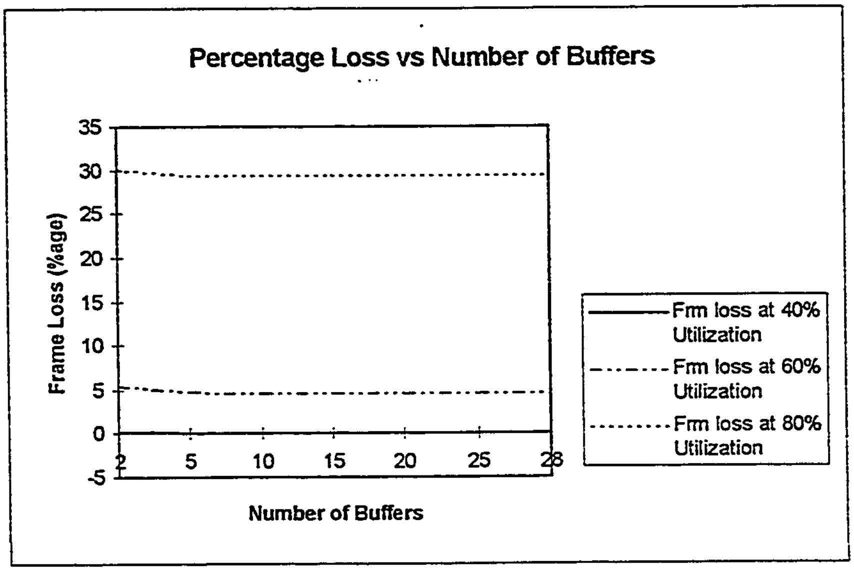


Figure 5-16 Frame loss versus number of buffers

Firewalling :

The firewalling capability of the bridge was verified against certain IP addresses. No packets for the blocked IP addresses were observed to pass through the bridge.

Statistics Collection and Display :

The bridge generates and broadcasts statistics bearing frames according to prescribed sample intervals. The statistics were satisfactorily received and displayed using the receiver program.

5 .4 Suggestions For Improvement

Choice of Network Interface card

As mentioned earlier, I/O mapped cards do not support DMA transfers and the CPU is required to copy the frame to the memory buffer thus keeping it busy during nearly all of the receiving period. A card supporting DMA could be used to alleviate this shortcoming. Further, it was mentioned that the *as_send_pkt ()* function can be used to queue frames to the card. If an interface card and packet driver are selected such that they support this function, then out-going frames may be queued to the card which will free the CPU for other tasks. This will improve the frame loss performance of the bridge.

Choice of a queuing strategy

If memory mapped interfaces are used, it is suggested to change the queuing scheme from a completely shared single queue to two queues per port. One for input frames and one for output frames. This will allow all the interfaces to receive and send frames independent of the main CPU. It will also provide for a scaleable design.

Separating the filtering and decision making from the receiver routine :

In the proposed design all the processing to decide whether the frame needs to be stored, discarded, or firewalled, is done by the receiver routine. These functions should be moved to the main program making the receiver routine short and efficient.

The above three suggestions will result in a bridge whose network interfaces are capable of receiving, storing and sending frames with minimal CPU intervention. Short and efficient receiving and sending routines will help to improve throughput and delay, because they will not keep the CPU in the "interrupts disabled" state for long. Choice of two queues (input and output) per port will help to alleviate the problem of buffer starvation. The main CPU will be available for the bridging and firewalling functions

CHAPTER 6

6 CONCLUSIONS

This chapter summarizes the work accomplished in this thesis and gives suggestions for further research.

6.1 Summary Of The Thesis

In this thesis, the packet driver interface specification standard was studied, and translated into functions which were then used to implement a working bridge. The various design issues were discussed and the choices made were presented. The bridge was tested and compared with other bridges available to us. Extensive experimental evaluation was conducted. The bridge exhibited very good performance under a variety of workload conditions. The main sources of performance degradation were identified and possible solutions were suggested. The bridge has two additional important capabilities: (i) Statistical data collection and, (ii) Firewalling capability. Next, we point out some of the non-conventional uses of this project and suggest some areas for future work.

6.2 Suggested Possible Applications

Apart from the basic bridging function, the bridge can be used to solve several other network problems without any modification (in some cases) or with a few minor modifications. Below, we enumerate several of these additional uses :

1) Uplink To Fast Ethernet Segment: The proposed bridge can be used as an inexpensive uplink to a fast Ethernet segment by installing one 100 Mbps network interface card and one 10 Mbps card and loading packet drivers for them. This use is not possible with other bridging programs such as the KarlBridge program which uses specific hardware and has drivers written for these cards for optimization. It is possible to use the proposed bridge in this configuration because it conforms to the packet driver specification standard, and nearly all cards have drivers written for this standard.

2) As Packet Filter Firewall: The bridge can be used as a packet filter to perform various filtering functions. The data filtering capability has been demonstrated in the previous chapter. Filtering out the various frames is only a matter of matching the right location in the frame with the right mask. Possible uses may be protocol filtering so as to limit visibility of servers and printers to one segment of a network, limit the usage of

certain software to a part of the network in order to enforce site licensing, restrict the usage of CD ROM databases, or secure systems to prevent unauthorized access. These secured systems may be subnetworks within the main network or it may be the main network itself. The main network may be secured from external users by limiting access to specific machines by placing the packet filter in front of the modem server.

3) As Network Monitor For The Segment: Some network interface cards can even provide statistics about erroneous frames, runts and stubs. As pointed out earlier the bridge can be used to generate and route statistics about usage and performance of various protocols and services which is helpful in planning and designing the network, or in testing new protocols.

4) Filtering for Network Performance Improvement: Bridges always forward multicast frames. Secondly, if the destination address is not found in the "learned address" table, the default is to forward it. Multicasting is generally not used except in rare cases. And where it is used, it (generally) uses higher layer multicasting features, rather than the frame level. Therefore, for a network that is not using Ethernet multicasting, it is possible to drop all multicast frames without any loss in functionality. The advantage of dropping Ethernet multicast frames can be explained as follows. If an Ethernet interface fails such that it sends out erroneous data. The frame produced in

such a situation will not only be forwarded[1] but it will also flood the "learned address" table with random source addresses. The bridge can be configured to drop all multicast addresses.

6.3 Suggested Further Research

The bridge designed and implemented in this work has a number of the basic functions that are common to other network systems. These functions can be used as a library to develop such systems. These functions are:

- Receive frames.
- Forward frames.
- Form and send frames.
- Interpret the contents of frames.
- Functions to initialize and control the interface
- Analyze and process frames based on some given decision.

Next, we suggest a number of possible improvements and extensions to the existing bridge capabilities. A few examples of future research projects are:

1. Internet packet filtering gateway: Implementing an Internet packet filtering gateway would require studying the various protocols used on the network and also the

---

[1] Because its destination address is random and will therefore be a multicast address fifty percent of the time and the rest of the time it will not be in the " address" table.

services provided by it, deciding on a policy, and formulating what is to be permitted. This bridge can be used as a building block.

2. A Central Network Monitoring and Analyzing Subsystem : The bridge collects segment statistics and broadcasts it periodically. The bridge also has the capability to route the statistics bearing datagram to a given IP address. Copies of the bridge can be spawned on Ethernet segments to collect local segment statistics and route it to a central location. The central facility can analyze and process this information to develop a network wide view.

3. Higher Level Gateways : Higher level relays like routers and gateways require the above mentioned functions as their basic functions. We believe that this work could form the basic building block for them.

4 Add The Spanning Tree Algorithm To the Bridge : The bridge conforms to IEEE's transparent bridge recommendation for Ethernet LANs. IEEE also recommends the use of the spanning tree algorithm to prevent frame loops. This algorithm can be added to the proposed bridge to make it capable of being deployed in LANs with multiple bridges.

7 APPENDIX A PACKET DRIVER SPECIFICATION

Contents

7 .1.1APPENDIX A1

PC/TCP Version 1.09 Packet Driver Specification

PC/TCP Packet Driver Specification

Revision 1.09
September-14-1989
Developed by:

FTP Software, Inc.
26 Princess St.
Wakefield, MA 01880-3004
(617) 246-0900

Support of a hardware interface, or mention of an interface manufacturer, by the Packet Driver specification does not necessarily indicate that the manufacturer endorses this specification.

1 Document Conventions

All numbers in this document are given in C-style representation. Decimal is expressed as 11, hexadecimal is expressed as 0x0B, octal is expressed as 013. All reference to network hardware addresses (source, destination and multicast) and demultiplexing information for the packet headers assumes they are represented as they would be in a MAC-level packet header being passed to the send_pkt() function.

2 Introduction and Motivation

This document describes the programming interface to FTP Software Packet Drivers. Packet drivers provide a simple, common programming interface that allows multiple applications to share a network interface at the data link level. The packet drivers demultiplex incoming packets among the applications by using the network media's standard packet type or service access point field(s).

The intent of this specification is to allow protocol stack implementations to be independent of the actual brand or model of the network interface in use on a particular machine. Different versions of various protocol stacks still must exist for different network media (Ethernet, 802.5 ring, serial lines), because of differences in protocol-to-physical address mapping, header formats, maximum transmission units (MTUs) and so forth.

The packet driver provides calls to initiate access to a specific packet type, to end access to it, to send a packet, to get statistics on the network interface and to get information about the interface.

Protocol implementations that use the packet driver can completely coexist on a PC and can make use of one another's services, whereas multiple applications which do not use the driver do not coexist on one machine properly. Through use of the packet driver, a user could run TCP/IP, XNS, and a proprietary protocol implementation such as DECnet, Banyan's, LifeNet's, Novell's or 3Com's without the difficulties associated with pre-empting the network interface.

Applications which use the packet driver can also run on new network hardware of the same class without being modified; only a new packet driver need be supplied.

Several levels of packet drivers are described in this specification. The first is the basic packet driver, which provides minimal functionality but should be simple to implement and which uses very few host resources. The basic driver provides operations to broadcast and receive packets. The second driver is the extended packet driver, which is a superset of the basic driver. The extended driver supports less commonly used functions of the network interface such as multicast, and also gathers statistics on use of the interface and makes these available to the application. The third level, the high-performance functions, support performance improvements and tuning.

Functions which are available in only the extended packet driver are noted as such in their descriptions. All basic packet driver functions are available in the extended driver. The high-performance functions may be available with either basic or extended drivers.

3 Identifying network interfaces

Network interfaces are named by a triplet of integers, <class, type,

number>. The first number is the class of interface. The class tells what kind of media the interface supports: DEC/Intel/Xerox (DIX or Bluebook) Ethernet, IEEE 802.3 Ethernet, IEEE 802.5 Token Ring, ProNET-10, Appletalk, serial line, etc.

The second number is the type of interface: this specifies a particular instance of an interface supporting a class of network medium. Interface types for Ethernet might name these interfaces: 3Com 3C503 or 3C505, Interlan NI5210, Univation, BICC Data Networks ISOLAN, Ungermann-Bass NIC, etc. Interface types for IEEE 802.5 might name these interfaces: IBM Token Ring adapter, Proteon p1340, etc.

The last number is the interface number. If a machine is equipped with more than one interface of a class and type, the interfaces must be numbered to distinguish between them.

An appendix details constants for classes and types. The class of an interface is an 8-bit integer, and its type is a 16 bit integer. Class and type constants are managed by FTP Software. Contact FTP to register a new class or type number.

The type 0xFFFF is a wildcard type which matches any interface in the specified class. It is unnecessary to wildcard interface numbers, as 0 will always correspond to the first interface of the specified class and type.

This specification has no provision for the support of multiple network interfaces (with similar or different characteristics) via a single Packet Driver and associated interrupt. We feel that this issue is best addressed by loading several Packet Drivers, one per interface, with different interrupts (although all may be included in a single TSR software module). Applications software must check the class and type returned from a driver_info() call in any case, to make sure that the Packet Driver is for the correct media and packet format. This can easily be generalized by searching for another Packet Driver if the first is not of the right kind.

4 Initiating driver operations

The packet driver is invoked via a software interrupt in the range 0x60 through 0x80. This document does not specify a particular interrupt, but describes a mechanism for locating which interrupt the driver uses. The interrupt must be configurable to avoid conflicts with other pieces of software that also use software interrupts. The program which installs the packet driver should provide some mechanism for the user to specify the interrupt.

The handler for the interrupt is assumed to start with 3 bytes of executable code; this can either be a 3-byte jump instruction, or a 2-byte jump followed by a NOP (do not specify "jmp short" unless you also specify an explicit NOP). This must be followed by the null-terminated ASCII text string "PKT DRVR". To find the interrupt being used by the driver, an application should scan through the handlers for vectors 0x60 through 0x80 until it finds one with the text string "PKT DRVR" in the 12 bytes immediately following the entry point.

5 Link-layer demultiplexing

If a network media standard is to support simultaneous use by different transport protocols (e.g. TCP/IP, XNS, OSI), it must define some link-level mechanism which allows a host to decide which protocol a packet is intended for. In DIX Ethernet, this is the 16-bit "ethertype" field immediately

following the 6-byte destination and source addresses. In IEEE 802.3 where 802.2 headers are used, this information is in the variable-length 802.2 header. In Proteon's ProNET-10, this is done via the 32-bit "type" field. Other media standards may demultiplex via a method of their own, or 802.2 headers as in 802.3.

Our access_type() function provides access to this link-layer demultiplexing. Each call establishes a destination for a particular type of link-layer packet, which remains in effect until release_type() is called with the handle returned by that particular access_type(). The link-layer demultiplexing information is passed via the type and typelen fields, but values and interpretation depend on the class of packet driver (and thus on the media in use).

A class 1 driver (DIX Ethernet) should expect type to point at an "ethertype" value (in network byte order, and greater than 0x05EE), and might reasonably require typelen to equal 2. A class 2 driver could require 4 bytes. However, a class 3 (802.5) or 11 (Ethernet with 802.2 headers) driver should be prepared for typelen values between 2 (for the DSAP/SSAP fields only) and 8 (3 byte 802.2 header plus 3-byte Sub-Network Access Protocol extension header plus 2-byte "ethertype" as defined in RFC 1042).

6 Programming interface

All functions are accessed via the software interrupt determined to be the driver's via the mechanism described earlier. On entry, register AH contains the code of the function desired.

The handle is an arbitrary integer value associated with each MAC-level demultiplexing type that has been established via the access_type call. Internally to the packet driver, it will probably be a pointer, or a table offset. The application calling the packet driver cannot depend on it assuming any particular range, or any other characteristics. In particular, if an application uses two or more packet drivers, handles returned by different drivers for the same or different types may have the same value.

Note that some of the functions defined below are labeled as extended driver functions and high-performance functions. Because these are not required for basic network operations, their implementation may be considered optional. Programs wishing to use these functions should use the driver_info() function to determine if they are available in a given packet driver.

6.1 Entry conditions

FTP Software applications which call the packet driver are coded in Microsoft C and assembly language. All necessary registers are saved by FTP's routines before the INT instruction to call the packet driver is executed. Our current receiver() functions behave as follows: DS, SS, SP and the flags are saved and restored. All other registers may be modified, and should be saved by the packet driver, if necessary. Processor interrupts may be enabled while in the upcall, but the upcall doesn't assume interrupts are disabled on entry. On entry, receiver() switches to a local stack, and switches back before returning.

Note that some older versions of PC/TCP may enable interrupts during the upcall, and leave them enabled on return to the Packet Driver.

When using a class 1 driver, PC/TCP will normally make 5 access_type() calls for IP, ARP and 3 kinds of Berkeley Trailer encapsulation packets. On other media, the number of handles we open will vary, but it is usually at least two (IP and ARP). Implementors should make their tables large enough to allow two protocol stacks to co-exist. We recommend support for at least 10 open handles simultaneously.

6.2 Byte and Bit ordering

Developers should note that, on many networks and protocol families, the byte-ordering of 16-bit quantities on the network is opposite to the native byte-order of the PC. (802.5 Token Ring is an exception). This means that DEC-Intel-Xerox ethertype values passed to access_type() must be swapped (passed in network order). The IEEE 802.3 length field needs similar handling, and care should be taken with packets passed to send_pkt(), so all fields are in the proper order. Developers working with MSB LANs (802.5 and FDDI) should be aware that a MAC address changes bit order depending on whether it appears in the header or as data.

6.3 driver_info()

```
        driver_info(handle)   AH == 1, AL == 255
                int     handle; BX              /* Optional */
```

error return:

```
        carry flag set
        error code              DH
        possible errors:
                BAD_HANDLE                      /* older drivers only */
```

non-error return:

```
        carry flag clear
        version                 BX
        class                   CH
        type                    DX
        number                  CL
        name                    DS:SI
        functionality           AL
                                    1 == basic functions present.
                                    2 == basic and extended present.
                                    5 == basic and high-performance.
                                    6 == basic, high-performance, extended.
                                    255 == not installed.
```

This function returns information about the interface. The version is assumed to be an internal hardware driver identifier. In earlier versions of this spec, the handle argument (which must have been obtained via access_type()) was required. It is now optional, but drivers developed according to versions of this spec previous to 1.07 may require it, so implementers should take care.

6.4 access_type()

```
        int access_type(if_class, if_type, if_number, type, typelen, receiver)
```

```
            AH == 2

            int       if_class;             AL
            int       if_type;              BX
            int       if_number;            DL
            char far *type;                 DS:SI
            unsigned typelen;               CX
            int       (far *receiver)();    ES:DI

error return:
        carry flag set
        error code                          DH
        possible errors:
            NO_CLASS
            NO_TYPE
            NO_NUMBER
            BAD_TYPE
            NO_SPACE
            TYPE_INUSE

non-error return:
        carry flag clear
        handle                              AX

receiver call:
        (*receiver)(handle, flag, len [, buffer])
            int       handle;               BX
            int       flag;                 AX
            unsigned len;                   CX
        if AX == 1,
            char far *buffer;               DS:SI
```

Initiates access to packets of the specified type. The argument type is a pointer to a packet type specification. The argument typelen is the length in bytes of the type field. The argument receiver is a pointer to a subroutine which is called when a packet is received. If the typelen argument is 0, this indicates that the caller wants to match all packets (match all requests may be refused by packet drivers developed to conform to versions of this spec previous to 1.07).

When a packet is received, receiver is called twice by the packet driver. The first time it is called to request a buffer from the application to copy the packet into. AX == 0 on this call. The application should return a pointer to the buffer in ES:DI. If the application has no buffers, it may return 0:0 in ES:DI, and the driver should throw away the packet and not perform the second call.

It is important that the packet length (CX) be valid on the AX == 0 call, so that the receiver can allocate a buffer of the proper size. This length (as well as the copy performed prior to the AX == 1 call) must include the MAC header and all received data, but not the trailing Frame Check Sequence (if any)..

On the second call, AX == 1. This call indicates that the copy has been completed, and the application may do as it wishes with the buffer. The buffer that the packet was copied into is pointed to by DS:SI.

6.5 release_type()

```
        int release_type(handle)        AH == 3
                int     handle;         BX

error return:
        carry flag set
        error code                      DH
        possible errors:
                BAD_HANDLE

non-error return:
        carry flag clear
```

This function ends access to packets associated with a handle returned by access_type(). The handle is no longer valid.

6.6 send_pkt()

```
        int send_pkt(buffer, length)    AH == 4
                char far *buffer;       DS:SI
                unsigned length;        CX

error return:
        carry flag set
        error code                      DH
        possible errors:
                CANT_SEND

non-error return:
        carry flag clear
```

Transmits length bytes of data, starting at buffer. The application must supply the entire packet, including local network headers. Any MAC or LLC information in use for packet demultiplexing (e.g. the DEC-Intel-Xerox Ethertype) must be filled in by the application as well. This cannot be performed by the driver, as no handle is specified in a call to the send_packet() function.

6.7 terminate()

```
        terminate(handle)          AH == 5
                int     handle;    BX

error return:
        carry flag set
        error code                 DH
        possible errors:
                BAD_HANDLE
                CANT_TERMINATE

non-error return:
        carry flag clear
```

Terminates the driver associated with handle. If possible, the driver will exit and allow MS-DOS to reclaim the memory it was using.

6.8 get_address()

```
        get_address(handle, buf, len)  AH == 6
                int     handle;        BX
                char far *buf;         ES:DI
                int     len;           CX

error return:
        carry flag set
        error code                     DH
        possible errors:
                BAD_HANDLE
                NO_SPACE

non-error return:
        carry flag clear
        length                         CX
```

Copies the current local net address of the interface into buf. The buffer buf is len bytes long. The actual number of bytes copied is returned in CX. If the NO_SPACE error is returned, this indicates that len was insufficient to hold the local net address. If the address has been changed by set_address(), the new address should be returned.

6.9 reset_interface()

```
        reset_interface(handle)         AH == 7
                int     handle;         BX
```

error return:

```
        carry flag set
        error code                      DH
        possible errors:
                BAD_HANDLE
                CANT_RESET
```

non-error return:

```
        carry flag clear
```

Resets the interface associated with handle to a known state, aborting any transmits in process and reinitializing the receiver. The local net address is reset to the default (from ROM), the multicast list is cleared, and the receive mode is set to 3 (own address & broadcasts). If multiple handles are open, these actions might seriously disrupt other applications using the interface, so CANT_RESET should be returned.

6.10 get_parameters()

high-performance driver function

```
        get_parameters()                AH = 10
```

error return:

```
        carry flag set
        error code                      DH
        possible errors:
                BAD_COMMAND             /* No high-performance support */
```

non error return:

```
     carry flag clear
     struct param far *;            ES:DI

  struct param {
     unsigned char  major_rev;      /* Revision of Packet Driver spec */
     unsigned char  minor_rev;      /*  this driver conforms to. */
     unsigned char  length;         /* Length of structure in bytes */
     unsigned char  addr_len;       /* Length of a MAC-layer address */
     unsigned short mtu;            /* MTU, including MAC headers */
     unsigned short multicast_aval; /* Buffer size for multicast addr */
     unsigned short rcv_bufs;       /* (# of back-to-back MTU rcvs) - 1 */
     unsigned short xmt_bufs;       /* (# of successive xmits) - 1 */
     unsigned short int_num;        /* Interrupt # to hook for post-EOI
                                       processing, 0 == none */
  };
```

The performance of an application may benefit from using get_parameters() to obtain a number of driver parameters. This function was added to v1.09 of this specification, and may not be implemented by all drivers (see driver_info()).

The major_rev and minor_rev fields are the major and minor revision numbers of the version of this specification the driver conforms to. For this document, major_rev is 1 and minor_rev is 9. The length field may be used to determine which values are valid, should a later revision of this specification add more values at the end of this structure. For this document, length is 14. The addr_len field is the length of a MAC address, in bytes. Note the param structure is assumed to be packed, such that these fields occupy four consecutive bytes of storage.

In the param structure, the mtu is the maximum MAC-level packet the driver can handle (on Ethernet this number is fixed, but it may vary on other media, e.g. 802.5 or FDDI). The multicast_aval field is the number of bytes required to store all the multicast addresses supported by any "perfect filter" mechanism in the hardware. Calling set_multicast_list() with its len argument equal to this value should not fail with a NO_SPACE error. A value of zero implies no multicast support.

The rcv_bufs and xmt_bufs indicate the number of back-to-back receives or transmits the card/driver combination can accommodate, minus 1. The application can use this information to adjust flow-control or transmit strategies. A single-buffered card (for example, an Interlan NI5010) would normally return 0 in both fields. A value of 0 in rcv_bufs could also be used by a driver author to indicate that the hardware has limitations which prevent it from receiving as fast as other systems can send, and to recommend that the upper-layer protocols invoke lock-step flow control to avoid packet loss.

The int_num field should be set to a hardware interrupt that the application can hook in order to perform interrupt-time protocol processing after the EOI has been sent to the 8259 interrupt controller and the card is ready for more interrupts. A value of zero indicates that there is no such interrupt. Any application hooking this interrupt and finding a non-zero value in the vector must pass the interrupt down the chain and wait for its predecessors to return before performing any processing or stack switches. Any driver which implements this function via a separate INT instruction and vector, instead of just using the hardware interrupt, must prevent any saved context from being overwritten by a later interrupt. In other words, if the driver switches to its own stack, it must allow re-entrancy.

6.11 as_send_pkt()

high-performance driver function

```
    int as_send_pkt(buffer, length, upcall) AH == 11
        char far *buffer;          DS:SI
        unsigned length;           CX
        int     (far *upcall)(); ES:DI
```

error return:

```
    carry flag set
    error code                     DH
    possible errors:

        CANT_SEND                  /* transmit error, re-entered, etc. */
        BAD_COMMAND                /* Level 0 or 1 driver */
```

non-error return:

```
    carry flag clear
```

buffer available upcall:

```
    (*upcall)(buffer, result)
        int     result;        AX     /* 0 for copy ok */
        char far *buffer;      ES:DI  /* from as_send_pkt() call */
```

as_send_pkt() differs from send_pkt() in that the upcall() routine is called when the application's data has been copied out of the buffer, and the application can safely modify or re-use the buffer. The driver may pass a non-zero error code to upcall() if the copy failed, or some other error was detected, otherwise it should indicate success, even if the packet hasn't actually been transmitted yet. Note that the buffer passed to send_pkt() is assumed to be modifiable when that call returns, whereas with as_send_pkt(), the buffer may be queued by the driver and dealt with later. If an error is returned on the initial call, the upcall will not be executed. This function was added in v1.09 of this specification, and may not be implemented by all drivers (see driver_info()).

6.12 set_rcv_mode()

extended driver function

```
        set_rcv_mode(handle, mode)  AH == 20
                int     handle;     BX
                int     mode;       CX
```

error return:

```
        carry flag set
        error code                  DH
        possible errors:
                BAD_HANDLE
                BAD_MODE
```

non-error return:

```
        carry flag clear
```

Sets the receive mode on the interface associated with handle. The following values are accepted for mode:

mode	meaning
1	turn off receiver
2	receive only packets sent to this interface
3	mode 2 plus broadcast packets
4	mode 3 plus limited multicast packets
5	mode 3 plus all multicast packets
6	all packets

Note that not all interfaces support all modes. The receive mode affects all packets received by this interface, not just packets associated with the handle argument. See the extended driver functions get_multicast_list() and set_multicast_list() for programming "perfect filters" to receive specific multicast addresses.

Note that mode 3 is the default, and if the set_rcv_mode() function is not implemented, mode 3 is assumed to be in force as long as any.handles are open (from the first access_type() to the last release_type()).

6.13 get_rcv_mode()

```
extended driver function
        get_rcv_mode(handle, mode) AH == 21
                int      handle;        BX

error return:
        carry flag set
        error code                      DH
        possible errors:
                BAD_HANDLE

non-error return:
        carry flag clear
        mode                            AX
```

Returns the current receive mode of the interface associated with handle.

6.14 set_multicast_list()

```
extended driver function
        set_multicast_list(addrlst, len)AH == 22
                char far *addrlst;      ES:DI
                int      len;           CX

error return:
        carry flag set
        error code                      DH
        possible errors:
                NO_MULTICAST
                NO_SPACE
                BAD_ADDRESS

non-error return:
        carry flag clear
```

The addrlst argument is assumed to point to an len-byte buffer containing a number of multicast addresses. BAD_ADDRESS is returned if len modulo the size of an address is not equal to 0, or the data is unacceptable for some reason. NO_SPACE is returned (and no addresses are set) if there are more addresses than the hardware supports directly.

The recommended procedure for setting multicast addresses is to issue a get_multicast_list(), copy the information to a local buffer, add any addresses desired, and issue a set_multicast_list(). This should be reversed when the application exits. If the set_multicast_list() fails due to NO_SPACE, use set_rcv_mode() to set mode 5 instead.

6.15 get_multicast_list()

extended driver function	
get_multicast_list()	AH == 23
error return:	
carry flag set	
error code	DH
possible errors:	
NO_MULTICAST	
NO_SPACE	
non-error return:	
carry flag clear	
char far *addrlst;	ES:DI
int len;	CX

On a successful return, addrlst points to len bytes of multicast addresses currently in use. The application program must not modify this information in-place. A NO_SPACE error indicates that there wasn't enough room for all active multicast addresses.

6.16 get_statistics()

extended driver function	
get_statistics(handle)	AH == 24
int handle;	BX
error return:	
carry flag set	
error code	DH
possible errors:	
BAD_HANDLE	
non-error return:	
carry flag clear	

```
        char far *stats;            DS:SI

    struct statistics {
        unsigned long  packets_in;    /* Totals across all handles */
        unsigned long  packets_out;
        unsigned long  bytes_in;      /* Including MAC headers */
        unsigned long  bytes_out;
        unsigned long  errors_in;     /* Totals across all error types */
        unsigned long  errors_out;
        unsigned long  packets_lost;  /* No buffer from receiver(), card */
                                      /*  out of resources, etc. */
    };
```

Returns a pointer to a statistics structure for the interface. The values are stored as to be normal 80xx 32-bit integers.

6.17 set_address()

```
extended driver function
        set_address(addr, len)      AH == 25
                char far *addr;     ES:DI
                int len;            CX

error return:
        carry flag set
        error code                  DH
        possible errors:
                CANT_SET
                BAD_ADDRESS

non-error return:
        carry flag clear
        length                      CX
```

This call is used when the application or protocol stack needs to use a specific LAN address. For instance, DECnet protocols on Ethernet encode the protocol address in the Ethernet address, requiring that it be set when the protocol stack is loaded. A BAD_ADDRESS error indicates that the Packet Driver doesn't like the len (too short or too long), or the data itself. Note that packet drivers should refuse to change the address (with a CANT_SET error) if more than one handle is open (lest it be changed out from under another protocol stack).

7 .1.2 APPENDIX A2

Packet Driver Specification
Interface classes and types

PC/TCP Version 1.09 Packet Driver Specification
FTP Software, Inc.

The following are defined as network interface classes, with their individual types listed immediately following the class.

DEC/Intel/Xerox "Bluebook" Ethernet

Class	1
3COM 3C500/3C501	1
3COM 3C505	2
Interlan Ni5010	3
BICC Data Networks 4110	4
BICC Data Networks 4117	5
MICOM-Interlan NP600	6
Ungermann-Bass PC-NIC	8
Univation NC-516	9
TRW PC-2000	10
Interlan Ni5210	11
3COM 3C503	12
3COM 3C523	13
Western Digital WD8003	14
Spider Systems S4	15
Torus Frame Level	16
10NET Communications	17
Gateway PC-bus	18
Gateway AT-bus	19
Gateway MCA-bus	20
IMC Pcnic	21
IMC PCnic II	22
IMC PCnic 8bit	23
Tigan Communications	24
Micromatic Research	25
Clarkson "Multiplexor"	26
D-Link 8-bit	27
D-Link 16-bit	28
D-Link PS/2	29
Research Machines 8	30

Research Machines 16	31
Research Machines MCA	32
Radix Microsys. EXM1 16-bit	33
Interlan Ni9210	34
Interlan Ni6510	35
Vestra LANMASTER 16-bit	36
Vestra LANMASTER 8-bit	37
Allied Telesis PC/XT/AT	38
Allied Telesis NEC PC-98	39
Allied Telesis Fujitsu FMR40	
Ungermann-Bass NIC/PS2	41
Tiara LANCard/E AT	42
Tiara LANCard/E MC	43
Tiara LANCard/E TP	44
Spider Comm. SpiderComm	845
Spider Comm. SpiderComm16	46
AT&T Starlan NAU	47
AT&T Starlan-10 NAU	48
AT&T Ethernet NAU	49
Intel smart card	50
ProNET-10	
Class	2
Proteon p1300	1
Proteon p1800	2
IEEE 802.5/ProNET-4	
Class	3
IBM Token ring adapter	1
Proteon p1340	2
Proteon p1344	3
Gateway PC-bus	4
Gateway AT-bus	5
Gateway MCA-bus	6
Omninet	
Class	4
Appletalk	
Class	5
Serial line	
Class	6
Clarkson 8250-SLIP	1

Clarkson "Multiplexor"	2	
Starlan		
Class	7	(NOTE: Class has been subsumed by Ethernet)
ArcNet		
Class	8	
Datapoint RIM	1	
AX.25Class	9	
KISS Class	10	
IEEE 802.3 w/802.2 hdrs		
Class	11	
Types as given under DIX Ethernet		
See Appendix D.		
FDDI w/802.2 hdrs		
Class	12	
Internet X.25		
Class	13	
Western Digital	1	
Frontier Technology	2	
N.T. LANSTAR (encapsulating DIX)		
Class	14	
NT LANSTAR/8	1	
NT LANSTAR/MC	2	

7 .1.3APPENDIX A3

Packet Driver Specification
Function call numbers

The following decimal numbers are used to specify which operation the packet driver should perform. The number is stored in register AH on call to the packet driver.

driver_info	1
access_type	2
release_type	3
send_pkt	4
terminate	5
get_address	6
reset_interface	7
+get_parameters	10
+as_send_pkt	11
*set_rcv_mode	20
*get_rcv_mode	21
*set_multicast_list	22
*get_multicast_list	23
*get_statistics	24
*set_address	25

+ indicates a high-performance packet driver function
* indicates an extended packet driver function

AH values from 128 through 255 (0x80 through 0xFF) are reserved for user-developed extensions to this specification. While FTP Software cannot support user extensions, we are willing to act as a clearing house for information about them. For more information, contact us.

7 .1.4 APPENDIX A4

Packet Driver Specification
Error codes

PC/TCP Version 1.09 Packet Driver Specification
FTP Software, Inc.

Packet driver calls indicate error by setting the carry flag on return. The error code is returned in register DH (a register not used to pass values to functions must be used to return the error code). The following error codes are defined:

1	BAD_HANDLE	Invalid handle number,
2	NO_CLASS	No interfaces of specified class found,
3	NO_TYPE	No interfaces of specified type found,
4	NO_NUMBER	No interfaces of specified number found,
5	BAD_TYPE	Bad packet type specified,
6	NO_MULTICAST	This interface does not support multicast,
7	CANT_TERMINATE	This packet driver cannot terminate,
8	BAD_MODE	An invalid receiver mode was specified,
9	NO_SPACE	Operation failed because of insufficient space,
10	TYPE_INUSE	The type had previously been accessed, and not released,
11	BAD_COMMAND	The command was out of range, or not implemented,
12	CANT_SEND	The packet couldn't be sent (usually hardware error),
13	CANT_SET	Hardware address couldn't be changed (more than 1 handle open),

14	BAD_ADDRESS	Hardware address has bad length or format,
15	CANT_RESET	Couldn't reset interface (more than 1 handle open).

7 .1.5 APPENDIX A5

Packet Driver Specification
802.3 vs. Blue Book Ethernet

PC/TCP Version 1.09 Packet Driver Specification D.1
FTP Software, Inc.

One weakness of the present specification is that there is no provision for simultaneous support of 802.3 and Blue Book (the old DEC-Intel-Xerox standard) Ethernet headers via a single Packet Driver (as defined by its interrupt). The problem is that the "ethertype" of Blue Book packets is in bytes 12 and 13 of the header, and in 802.3 the corresponding bytes are interpreted as a length. In 802.3, the field which would appear to be most useful to begin the type check in is the 802.2 header, starting at byte 14. This is only a problem on Ethernet and variants (e.g. Starlan), where 802.3 headers and Blue Book headers are likely to need co-exist for many years to come.

One solution is to redefine class 1 as Blue Book Ethernet, and define a parallel class for 802.3 with 802.2 packet headers. This requires that a 2nd Packet Driver (as defined by its interrupt) be implemented where it is necessary to handle both kinds of packets, although they could both be part of the same TSR module.

As of v1.07 of this specification, class 11 was assigned to 802.3 using 802.2 headers, to implement the above.

Note: According to this scheme, an application wishing to receive IP encapsulated with an 802.2 SNAP header and "ethertype" of 0x800, per RFC 1042, would specify an typelen argument of 8, and type would point to:

```
char    iee_ip[] = {0xAA, 0xAA, 3, 0, 0, 0, 0x00, 0x08};
```

James B. VanBokkelen
jbvb@ftp.com
...!ftp!jbvb

8 BIBLIOGRAPHY

[1] Floyd Backes, "Transparent Bridges for Interconnection of IEEE 802 LANs", IEEE Network. Vol. 2, No. 1., pp. 5 - 9, Jan. 1988.

[2] Dough Karl, "Kbridge.txt", Unpublished, OSU. 1993.

[3] Gerd E. Keiser, *Local Area Networks*, McGraw Hill Book, Inc. 1989.

[4] "Packet Driver Specification", Internet Tools, Info Magic, Sept. 1994. (site dumps on CDROM).

[5] Gary R. McClain (editor), *The Handbook of International Connectivity Standards*, Multiscience Press, Inc. 1992.

[6] Mark A. Miller, P.E., *LAN Protocol Handbook*, M&T Publishing, Inc. 1990.

[7] Mark A. Miller, *Lan TroubleShooting Handbook*, M&T Publishing, Inc. 1989.

[8] Sharon Fisher, "Mix-and-Match Network Adapters", Byte Magazine, pp. 277-279, August 1990.

[9] John Romkey, Sharon Fisher, "Packet Drivers", Byte Magazine, pp. 297-306, May 1991.

[10] William Stallings, *Data and Computer Communications, Fourth Edition*, Prentice Hall International Editions. McMillan Publishing Co. 1994.

[11] Bill Hancock, *Designing and Implementing Ethernet Networks*, Second Edition, Bill Hancock and Essential Resources, Inc., 1988.

[12] Joseph L. Hammond, Peter J. P. O'Reilly, *Performance Analysis of Local Computer Networks*, Addison-Wesley Publishing Company, Inc. 1986.

[13] Andrew S. Tanenbaum, *Computer Networks*, Second Edition, Prentice-Hall International Editions. 1989.

[14] Dimitri Bertsekas, Robert Gallager, *Data Networks (Second Edition)*, Prentice-Hall International, Inc. 1992.

[15] William R. Cheswick, Steven M. Bellovin, *Firewalls and Internet Security : Repelling the Wily Hacker*, Addison-Wesley Publishing Company. 1994.

[16] Stephen Kent, "Comments on "Security Problems in the TCP/IP Protocol Suite"", Computer Communications Review, ACM SIGCOMM. pp. 10-19.

[17] S. M. Bellovin, "Security Problems in the TCP/IP Protocol Suite", Computer Communications Review, ACM SIGCOMM. pp. 32-48.

[18] R. T. Morris, "A Weakness in the 4.2BSD UNIX TCP/IP Software", Computing Science Technical Report No. 117, AT&T Bell Laboratories, Murray Hill, New Jersey. 1985.

[19] S. Bellovin, "Firewall - Friendly FTP", Internet Request for Comments No. 1579 (RFC1579). Feb. 1994.

[20] Douglas E. Comer, *Internetworking with TCP/IP* , Volume 1, Prentice Hall, 1991.

9 VITA

Amir Amanullah Khan.

Born at Karachi, Pakistan in 1962.

Received Bachelor's degree in Electrical Engineering from the University of Engineering and Technology, Lahore, Pakistan in February 1986.

Completed Master's degree requirements at King Fahad University of Petroleum and Minerals, Dhahran, Saudi Arabia in June 1996.