

MANAGING THE BANDWIDTH
OF
JNU CAMPUS WIDE NETWORK

*Dissertation submitted to Jawaharlal Nehru University in partial fulfillment of the
requirements for the award of the degree of*

Master of Technology

in

Computer Science and Technology

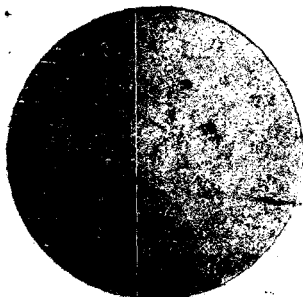
By

CHALLA BHANU PRASAD



**SCHOOL OF COMPUTER & SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110067**

JANUARY 2001






जवाहरलाल नेहरू विश्वविद्यालय
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI-110067

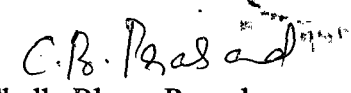
SCHOOL OF COMPUTER & SYSTEMS SCIENCES


CERTIFICATE

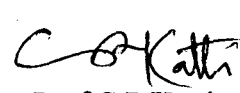
This is to certify that the dissertation entitled Managing the Bandwidth of JNU Campus Wide Network being submitted by CHALLA BHANU PRASAD to the School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi in partial fulfillment of the requirements for the award of the degree of Master of Technology in Computer Science is a bonafide work carried out by him under the guidance and supervision of Prof.G.V.Singh and Dr.D.K.Lobiyal.

The matter embodied in the dissertation has not been submitted for the award of any other degree or diploma.


Prof. G.V. Singh,
Professor, SC&SS,
Jawaharlal Nehru University,
New Delhi - 110067.


Challa Bhanu Prasad


Dr. D.K. Lobiyal,
Asst. Professor, SC&SS,
Jawaharlal Nehru University,
New Delhi - 110067.


Prof. C.P. Katti,
Dean, SC&SS,
Jawaharlal Nehru University,
New Delhi - 110067

***...dedicated to
my beloved parents***

ACKNOWLEDGEMENTS

I would like to pay obeisance at the feet of my beloved parents for their blessings are always with me in all my aspirations including my academics.

I would like to sincerely thank my supervisors, Prof. G.V. Singh and Dr. D. K. Lobiyal, School of Computer and Systems Sciences, Jawaharlal Nehru University for their help, encouragement and support extended in completion of this project. The valuable discussions were very much helpful in keeping the project in the right track.

I would like to record my sincere thanks to my dean Prof. C. P. Katti and Communication and Information Services, Jawaharlal Nehru University for providing the necessary computing facilities.

I take this opportunity to thank all of my faculty members and friends for their help and suggestions during the course of my project work.

Challa Bhanu Prasad

ABSTRACT

The assignment of appropriate data carrying capacity within the network for every user and application is the ultimate goal of monitoring and managing the bandwidth of an Internet edge link. Such a link could be, for example, a campus Internet access link or a small ISP's backbone access link.

We use packet monitoring to obtain traffic statistics, and blocking TCP requests for bandwidth control. Packet capturing implies capturing all packets that travel in the network. This can be achieved in an Ethernet Network because of its broadcasting nature. That is any packet sent over an Ethernet network is broadcasted to all machines in the network. The Ethernet card in every machine checks whether the particular packet is destined to it, if so accepts it else rejects it. This basic functionality of an Ethernet network is exploited to capture all packets that travel through the network.

Blocking a TCP connection and balancing the load on the network can be achieved by sending a segment that has the RST bit set to any one side of the connection. A monitoring and control architecture has been proposed to carry out the above mentioned operations. Further enhancements that manage the bandwidth of the campus wide network have been discussed.

TABLE OF CONTENTS

1. INTRODUCTION	1
2. BANDWIDTH MANAGEMENT	5
2.1 Scope of the Bandwidth Management	5
2.1.1 Characteristics of Bandwidth-Management	6
2.2 Motivations for Bandwidth Management	7
2.2.1 Necessity of Bandwidth Managment	7
3. MONITORING AND CONTROL ARCHITECTURE	9
3.1 The Architecture	9
3.2 Load Balancing	12
3.3 Structure of the Driver used for Capturing Packets on the Network	12
3.4 The Filtering Process	15
3.5 The Reading and Buffering Processes	15
3.6 The Writing Process	17
3.7 Statistics Mode	17
3.8 Data Structures and Functions	18
4. PACKET MONITORING	29
4.1 Modes in an Ethernet Interface Card	29
4.2 Sample PACKET.SYS Driver	30
4.3 TCP/IP Layered Architecture	37
4.4 Identifying Users Different Requests	38
4.4.1 Identifying a TCP/IP Packet	40
4.4.2 Identifying a HTTP Request	42

5. NETWORK BANDWIDTH CONTROL	44
5.1 TCP Communication in a network	44
5.2 Establishing a TCP connection	46
5.3 TCP connection Reset	46
5.4 Breaking a TCP connection	46
5.5 Framing a RST packet	47
CONCLUSION	57
REFERENCES	58

CHAPTER 1

INTRODUCTION

This chapter gives a brief overview of the Internet as mesh of interconnected users and service providers, explaining how service providers operate, and some examples of policies that can be set up based on Internet access link utilization by the network manager.

A network user is a person using a computer which is connected to a network, for accessing services delivered over that network such as e-mail. An internet is simply a collection of networks which are interconnected by links of various kinds. One special internet is the Internet, which is global internet of connected networks which all use the TCP/IP (Transmission Control Protocol / Internet Protocol) suit of network protocols.

The network to which an individual host is attached may be provided by her organization – a campus network or by a public Internet Service Provider (ISP). In either case, the network provides transport for the packets which carry the users data.

Services such as e-mail, news and the world wide web (WWW) are based on *server hosts*. Such servers may be operated by ISP, this is usually the case for e-mail, network news and chat services. Other servers, such as web servers and information or e-commerce servers and the services which are based on them are often operated by independent providers known as *Content Providers*.

The information between user hosts and services is carried in *packets*, i.e., short messages (typically containing 64 to about 1500 bytes) which travel across the Internet. A stream of packets between a particular host and server is often described as a flow of

network traffic; if we wish to know how the Internet is being used, we will have to measure these traffic flows.

Problem Definition

Managing the bandwidth of JNU Internet edge link with a view towards certain quality of service objectives for the services it carries.

Ordinary users connect to the Internet via the public telephone network, using modems to connect to their Internet Service Provider (ISP). We call this kind of ISP as Local ISP. A Local ISP makes it possible for individual users to attach their host computers to the Local ISP's network, and thus to the global Internet.

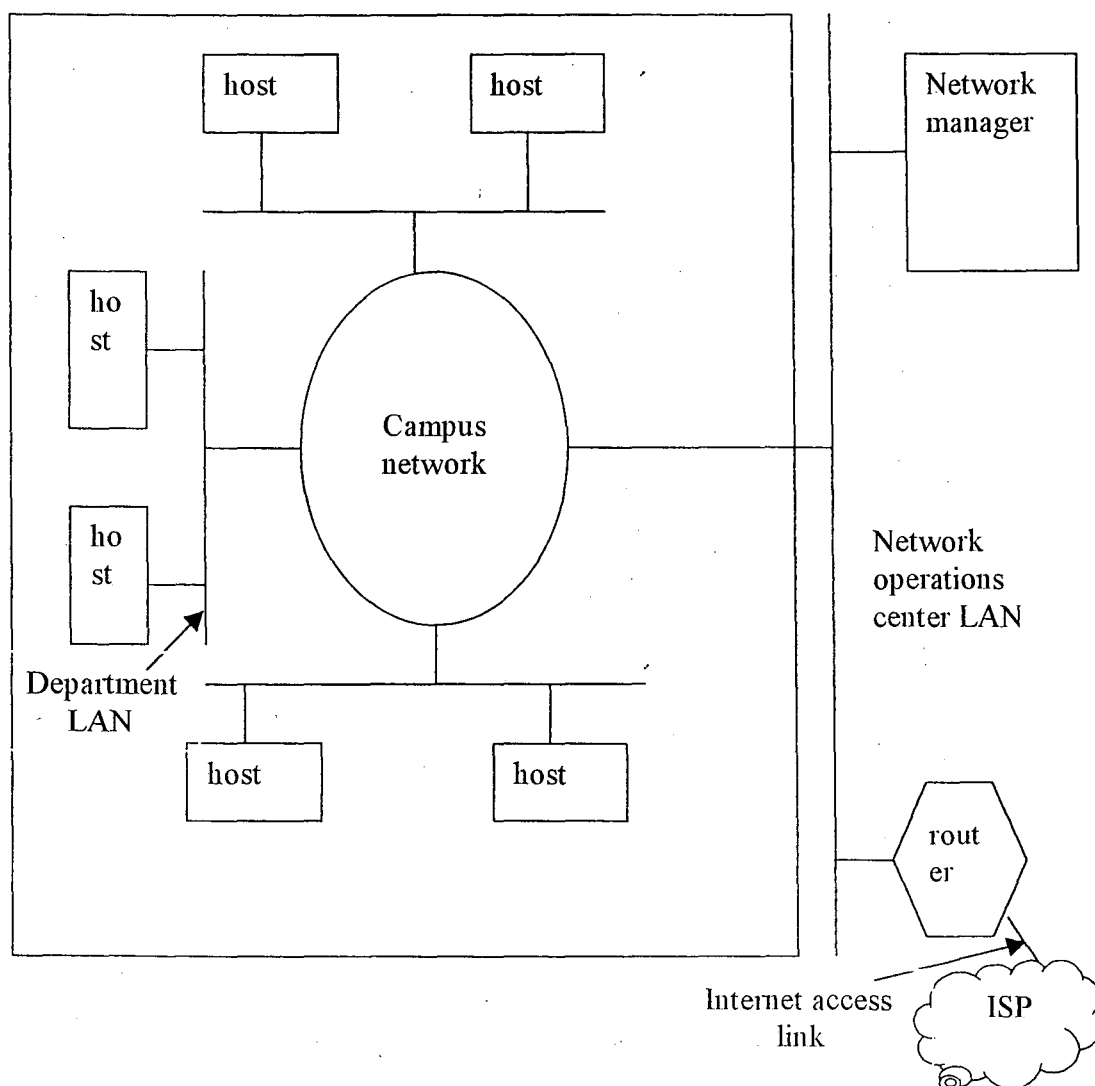


Figure: 1 A campus network with an access link to an Internet Service Provider

Packets arriving at a router are sent on towards their destination along the best possible path. This path is other networks in the Internet. This kind of transmission is described as *connectionless packet routing*. Until very recently all packets within the Internet were forwarded using the best possible path at each step in their paths; this is described as *best effort* service. Protocols for other kinds of service are beginning to emerge, for example the IETF's differentiated service (diffserv) protocol allows for various levels of priority traffic, and so on. The ability to offer services other than best effort will provide significant new opportunities for ISPs over the next few years.

Figure 1 shows a campus wide network and its attachment to an Internet Service Provider (ISP). Typical campus backbone speeds are 10 to 100Mbps, with more recent installations at 155Mbps if based on asynchronous transfer mode (ATM) or 1000Mbps if based on Gigabit Ethernet. Campus end points are typically connected to the backbone by departmental LANs of at least 10Mbps. As opposed to these numbers, Internet access link speeds range from 64Kbps to 2Mbps (our network link speed being 512Kbps). The access links with 2Mbps are fairly common now in developed countries, but still quite expensive and uncommon in developing countries. It is immediately clear from the scenario described above, that the WAN access link can become a bottleneck resource that needs to be monitored and managed.

All traffic between the access link and the campus network is made to pass through an Ethernet segment or, equivalently, an Ethernet hub. On this segment resides a machine called the network manager, with a module required to monitor and control the network.

The packet capturing is based on the utilities for promiscuously capturing IP packets on the Ethernet segment. The captured packet data is used to generate source-destination statistics and service statistics (like e-mail, HTTP, FTP) for the packets flowing on the Internet access link.

Based on the link utilization policies set up by the network manager, the packet monitoring module generates the traffic statistics. The bandwidth control module then controls the TCP connections to control the load on the Internet access link so that the traffic flowing on the link conforms to the configured policies. Some examples of policies that can be set up:

- Do not allowing the network users to access a set of (IP addresses) that are obviously very unproductive there by utilizing the network resources efficiently.
- Do not allowing the occupancy of the access link to exceed 90%; this threshold may be obtained, for example, from a model of TCP connections over a WAN with this link as the bottleneck link; the objective is to meet a session throughput objective.

There are several advantages to our approach:

1. Efficient utilization of the bandwidth capacity of the campus wide Internet access link.
2. QoS (Integrated Services) will be provided to the users of the network.
3. Easy installation in the network (most WAN access links are attached to campus network via a router on an Ethernet LAN. Installation of a system based on our approach is then just a matter of installing one or more machines on the same LAN).
4. Network unaffected by the manager hardware failure (in our approach, if the manager machine fails, the controls stop working, but the network traffic continues to flow, albeit in the original uncontrolled manner).

In this dissertation, we describe bandwidth management, proposed monitoring and control architecture and the algorithms that could help in implementing the work.

CHAPTER 2

BANDWIDTH MANAGEMENT

Bandwidth is the amount of data that can be transmitted in a fixed amount of time. For digital devices, the bandwidth is usually expressed in bits per second (bps or bytes per second). For analog devices, the bandwidth is expressed in cycles per second, or Hertz (Hz).

The ultimate goal of bandwidth management is the assignment of appropriate carrying capacity within the network for every user and application.

2.1 Scope of the Bandwidth Management

Today's best-effort model for public and private networks is based on the ideas that traffic is created equal. By contrast, network services such as the public switched telephone network have separate channels for signaling traffic and voice traffic, and they guarantee a fixed capacity end to end with small, predictable delays.

As we rely on best-effort networks to carry concurrent voice, video, data, and interactive application across a common network infrastructure, we must offer each of these traffic types the specific services and handling characteristics it requires.

Managing bandwidth means controlling traffic levels in real-time in order to make optimal use of scarce capacity.

2.1.1 Characteristics of a Bandwidth-managed Network

As described above the ultimate goal of bandwidth management is the assignment of appropriate carrying capacity within the network for every user and application. This statement can be further explained as:

“appropriate” capacity means the right data rate, the proper delay, and the right level of change in delay.

“with in the network” means that a network has differentiated services to offer. If a network is to provide more than one sort of service, then it must be able to identify different types of traffic and handle them in different ways.

“for every user and application” implies that the network has user-based and application-based identification systems. Today’s networking devices work with addresses and interfaces, so a bandwidth-managed network needs to determine user and application information from addressing information using a variety of network services such as user-based logins. This represents a significant shift away from a low-level addressing perspective and towards a view of the network based on users and services as the networking infrastructure starts to offer so-called “policy” capabilities.

A bandwidth-managed network is able to offer distinct characteristics based on the kinds of traffic it handles. Traffic can be classified by application, user, or even “external” factors such as time-of-day, date, or level of network congestion. While today’s technology won’t enable us to build a perfectly manageable network, we can take steps to build a network that can offer varying degrees of service and to deploy the system of directories and authentication points that will govern allocation of these services. Managing bandwidth properly will postpone the needs for additional bandwidth, making it a suitable alternative to increased LAN capacity.

Typically a network manager changes network behavior by altering parameters on a device manually, using a management console. Such a change is called an “administrative configuration” if the change is repeated across a range of devices automatically and it involves the creation of specific services, then the process is known as “provisioning”. A simple association of application types with relative priorities that can be configured administratively on an individual device or provisioned for a specific service can administer some degree of Quality of Service.

Because network bandwidth is a scarce resource, devices can be said to enforce the use of networked capacity. Enforcers attempt to deliver optimal bandwidth allocation, delay, variance in delay, and so on. Enforcing devices can send different kinds of traffic across different paths in the networks or they can treat traffic differently as it passes through them, using packet filters, firewalls etc.

2.2 Motivations for Bandwidth Management

If we’re going to invest time and effort in the creation of a differentiated-service network, there had better be good reasons to do so. Fortunately, there are many such reasons. In this section, we explore the motivations for bandwidth management, and look at the benefits and drawbacks of so-called “managed” bandwidth versus the “big” bandwidth approach of simply throwing capacity at congestion problems.

2.2.1 Necessity of Bandwidth Management

A fundamental question an IT manager must ask before embarking on the deployment of a bandwidth-managed network is whether the total cost of managing the network is lower than the cost of throwing bandwidth at congestion problems.

Managing bandwidth in detail is an extremely complicated task. Handling the myriad requests for response time, jitter, and capacity with dimensions of application, user, time-of-day, congestion, and link type is so complicated, in fact, that it’s *always*

cheaper to throw faster boxes at the problem on the LAN and leave it to someone else on the WAN.

Throwing bandwidth at the problem is naive. It discounts the impact of network failures, the delays in deploying capacity broadly, and the effects of application bandwidth hunger in coming years. Without controls, even a network of enormous bandwidth can still be overrun by miscreant applications. If a network cannot be throttled to some degree, managers cannot ensure the proper service of mission-critical applications. But the reality is that it's acceptable to throw bandwidth at a problem when it is cost effective to do so.

CHAPTER 3

MONITORING AND CONTROL ARCHITECTURE

Figure 1 shows the placement of the monitoring and control device on the Ethernet segment. All traffic between the bottleneck access link and the campus network is made to go through an Ethernet segment on which the machine is placed.

3.1 The Architecture

A block diagram shown in Figure 2 describes how the packets that come to the NIC can be captured, and how packet with RST bit set is sent on to the network if the source IP address of the packet matches any of the address in the blocking file. The machine with such an IP address is restricted from accessing the Internet.

The module comprises two threads one for grabbing the packets from the network and another for performing matching and sending the reset packets.

To perform a blocking operation, after obtaining the packet we have to compare it with the addresses that are specified in the blocking file, frame the reset packet and send it to the machine that has to be restricted. These operations put together will be very slow when compared to the rate at which the packets come to the NIC. If the application is a single-threaded application, which will read a packet and perform the above operations and then read the next packet, it is sure of losing packets. To prevent this we go in for a multi-threaded application with one thread continuously reading packets and the other

doing the blocking. The application has to make a ReadFile call to read a packet that comes to the NIC. Once the driver reads a packet from the NIC, while application is busy in performing other activity, the driver has to copy its buffer to the buffer of the application in other words, the packet has to be lifted up from the kernel level to the user

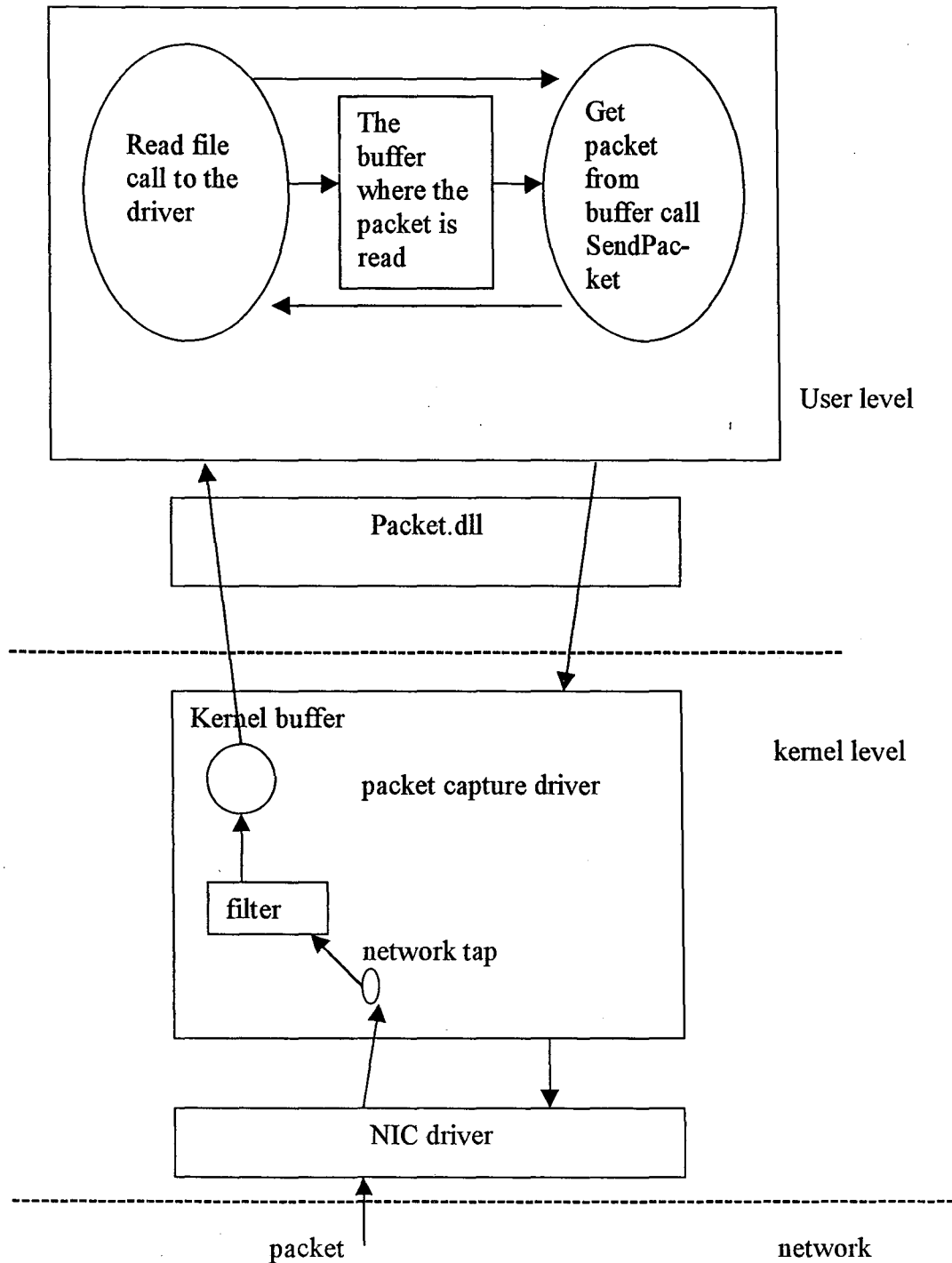


Figure: 2 Monitoring and Control Architecture

level. The lifting process takes quite sometime, if with in this time a packet comes to the NIC that packet is not grabbed and is lost.

Thus even if we call the ReadFile function continuously in an infinite while loop we tend to lose packets. This is because only after one ReadFile returns the other is initiated and so on. Thus if during a ReadFile is in progress a packet comes, the packet is lost. To avoid this, instead of making the ReadFile call synchronous, we make this asynchronous. By asynchronous, we mean that it is not necessary to wait until the previous ReadFile call returns, instead we can initiate some more reads. Let us see with this how we overcome the problem of missing the packets that come during the ReadFile is in progress. Consider that we have initiated 15 reads, now when a packet comes, the first read is invoked and it keeps reading. During this time if another packet comes then the second read that we have initiated gets invoked and similarly the other reads that we have initiated. Once a read is completed the blocking thread gets notified and this thread gets the buffer in which the packet has been read. The thread matches the obtained packet's IP address with those specified in the blocking file and then calls the SendPacket routine that we have already discussed to send a reset packet. Once this is over, the blocking thread indicates the grabber thread that initiates another read on this buffer. Thus at any time there will be a pending read and if any packet comes to the NIC it will be automatically grabbed. Even after this, if the rate at which the packets come to the NIC is very high then there are possibilities of missing the packets. But doing asynchronous reads gives the maximum efficiency in the sense that losing packets has the least probability.

1. Once a ReadFile returns the grabber thread indicates the completion of the read to the blocker thread. The blocker waits on an IO completion port for the Read to complete. Once the read is over the buffer is presented to the blocker thread. The blocker thread then compares the IP addresses and if it matches it calls the SendPacket to send a reset packet to the machine that has to be blocked.

2. Indicates that the Grabber thread fills the buffer with the packet that it has picked up

from the network.

3. Indicates that the blocker thread reads the packet that is placed by the grabber in the buffer.

4. Once the blocker sends a reset packet to the machine or finishes comparing the addresses, it posts a message with the buffer to the grabber thread. Once the grabber thread receives this message and the buffer, it initiates another ReadFile call on this buffer. Thus the application continuously keeps grabbing all the packets that come to the network and will not allow the machines that are added in the blocking file from accessing the Internet. To enter the IP addresses of the machines that need to be blocked, into the blocking file, the user is provided with a simple UI to do so.

3.2 Load Balancing

Similarly we can initiate a thread in the monitoring application that increments the count of the number of packets whenever a packets comes to its NIC. If we want to block the new connections when the load on the network exceeds the threshold value then what we can do is specify another file in which we will enter IP addresses of the sites that have to be restricted. Once a packet is obtained, the blocker thread compares the source IP address and the destination IP address with the set of addresses in the blocking file and act accordingly.

Since we do not use any queue or other variables that are accessed by both the threads simultaneously we have not dealt with the thread synchronization issues. If a queue or some other data structure is used that is accessed by both the threads, then the accesses have to be synchronized using critical section variables or other thread synchronization mechanisms.

3.3 Structure of the driver used for capturing packets on the network

The packet capture driver adds to windows kernels the capability to capture raw packets from a network in a way very similar to UNIX kernels with BPF. In addition, it

provides some functions, not available in the original BPF driver, to help the development of network test and monitor programs. Main goals of the packet capture driver are high capture performance, flexibility and compatibility with the original BPF for UNIX. This results into a protocol driver that can:

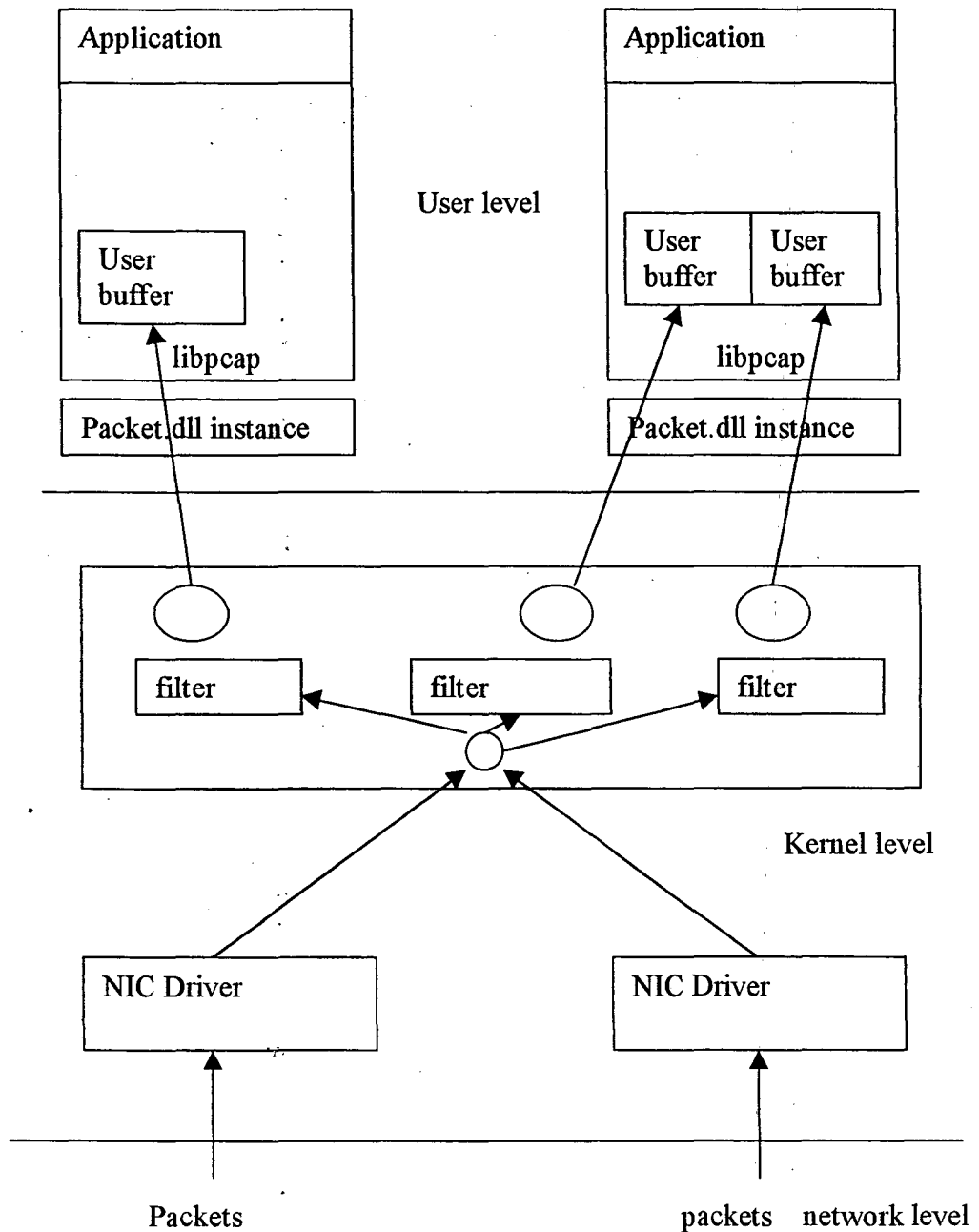


Figure: 3 structure of the driver with two adapters and a two applications

1. capture the raw traffic from the network and pass it to a user level application, and filter the incoming packets executing BPF pseudo-machine code. This means that the capture application can define a standard BPF program and pass it to the driver. The driver will discard the incoming packets that do not satisfy the filter, holds the packets in a buffer when the application is busy or it is not fast enough to sustain the flow of packets coming from the network, and collects the data from several packets and return it as a unit when the application does a read. To maintain packet boundaries packets are encapsulated in a header (the same used by BPF) that includes a time stamp, length, and offsets for data alignment.

2. write raw packets to the network calculate statistics on the network traffic

The real structure is more complex and can be seen in figure 3. This figure shows the driver's configuration with two network adapters and two capture applications.

For each capture session established between an adapter and a capture program, the driver maintains a filter and a buffer. The single network interface can be used by more than one application at the same time. For example, a user that wants to capture the IP and the UDP traffic and save them in two separate files, can launch two sessions of WinDump on the same adapter (but with different filters) at the same time. The first session will set a filter for the IP packets (and a buffer to store them), and the second a filter for the UDP packets. It is also possible to write an application that, interacting with the capture driver, is able to receive packets from more than one interface at the same time.

The interaction with NDIS is very similar under the different platforms and is obtained by a set of callback functions exported by the driver and a group NDIS library functions (*NdisTransferData*, *NdisSend...*) used by the packet driver to communicate with the NIC driver. Differences between the different flavors concerns the interaction with the other parts of the operating system (read and write call handling from user-level

applications, timer functions...), since the philosophy of the various operating systems is quite different.

3.4 The filtering process

The filter mechanism present in the packet capture driver derives directly from the BPF filter in UNIX. An application that needs to set a filter on the incoming packets can build a standard BPF filter program (for example through a call to the *pcap_compile* function of libpcap) and pass it to the driver, and the filtering process will be done at kernel level. The BPF program is transferred to the driver through an IOCTL call with the control code set to *pBIOCSETF*. A very important thing is that the driver needs to be able to verify the application's filter code. If no filter is defined, the driver accepts all the incoming packets.

The filter is applied to a packet when it's still in the NIC driver's memory, without copying it to the packet capture driver. This allows to reject a packet before any copy thus minimizing the load on the system.

3.5 The reading and buffering processes

When an application wants to obtain the packets from the network, it performs a *read* call on the NDIS packet capture driver (this is not true in Windows 95, where the application retrieves the data through a IOCTL call; however the result is the same). This call can be synchronous or asynchronous, because the driver offers both the possibilities. In the first case, the read call is blocking and the application is stopped until a packet arrives to the machine. In the second case, the application is not stopped and must check when the packet arrives. The usual and RECOMMENDED method to access the driver is the synchronous one because of the difficulties in implementing the asynchronous one that can bring to errors. The asynchronous method can be used to do application-level buffering, but this is usually not needed because the buffering in the driver is more efficient and clean.

After an incoming packet is accepted by the filter, the driver can be in two different situations:

The application is ready to get the packet, i.e. it executed a read call and is currently asleep waiting for the result of the call. In this case the incoming packet is immediately copied in the application's memory, the read call is completed and the application is waked up.

There is no pending read at the moment, i.e. the application is doing other stuff and is not blocked waiting for a packet. To avoid the loss of the packet, it must be stored in the driver's buffer and transferred to the application when it will be ready to receive it, i.e. at the next read call.

The driver uses a circular buffer to store the packets. A packet is stored in the buffer with a header that maintains information like the timestamp and the size of the packet. Incoming packets are discarded by the driver if the buffer is full when a new packet arrives. In fact, it is likely that a capture application, that needs to make operations on each packet, sharing at the same time the processor with other tasks, will not be able to work at network speed during heavy traffic or bursts. This problem is more noticeable on slower machines. The driver, on the other hand, runs at kernel level and is written explicitly to capture the packets, so it is very fast and usually it does not lose packets. Therefore an adequate buffer in the driver can store the packets while the application is busy, can compensate the slowness of the application and can avoid the loss of packets during bursts or high network activity.

If the buffer is not empty when the application performs a read system call, the packets in the driver's buffer are copied to the application memory and the read call is immediately completed. More than one packet can be copied from the driver's circular buffer to the application with a single read call. This improves the performances because it minimizes the number of reads.

The packet driver handles the kernel-level and user-level packet buffers in a very versatile way. It is possible to choose any dimension for the driver's circular buffer, limited only by the RAM of the machine. Furthermore, also the buffer used by the user-level application can have any size and can be changed whenever the application needs to do so. An important feature is that the two buffers are not forced to have the same size.

The packet capture driver detects the dimension of the application's buffer and fills it in the right way even when it has different size from the circular buffer. This is important when the driver's buffers is big, because a large application buffer would be a waste of memory. On the other hand, this mechanism has also a drawback: since the size of the packets is not fixed, when the dimension of the application's buffer is smaller than the number of bytes in the driver's buffer, it can happen that the driver has to scan the headers of the packets in its buffer in order to determine the amount of bytes to copy. This process slows down a bit the capture process, therefore best performance are obtained when application and kernel buffers have the same size.

3.6 The writing process

The packet capture driver allows to write raw data to the network. This can be useful to test either the network or the protocols and applications working on it. To send data, a user-level application performs a write system call on the packet driver's device file. The data is sent to the network as is, without encapsulating it in any protocol, therefore the application will have to build the various headers for each packet. The application does not need to generate the FCS because it is calculated by the network adapter hardware and it is attached automatically at the end of a packet before sending it to the network.

3.7 Statistics mode

A network manager is often not interested in the whole sequence of packets transiting on a network, but in statistical values about them. For example the user could be interested in network's utilization, broadcast level, but also in more refined information, like amount of mail traffic or number of web requests per second. 'Statistics' mode, or *mode 1*, is a particular working mode of the BPF capture driver that can be used to perform real time statistics on the network's traffic in an easy way and with the minimum impact on the system.

To put the driver in statistics mode, the user-level application makes an IOCTL call with code `pBIOCSMODE`, and the value '1' as input parameter. To set again the normal 'capture' working mode the same IOCTL, but with '0' as input parameter, can be called.

When in mode 1, the driver does not capture anything but limits itself to count the number of packets and the amount of bytes that satisfy the user-defined BPF filter. These values are passed to the application at regular intervals, whenever a timeout expires. The default value of this timeout is 1 second, but it can be set to any other value (with a 1 ms precision) with an IOCTL call (parameter `pBIOCSRTIMEOUT`). The counters are encapsulated in a `bpf_hdr` structure before being passed to the application.

In Windows NT/2000 the packet driver is seen by the applications as a network service. This means that an application uses the driver like a normal file: it reads and writes data from the network using the *readfile* and *writefile* system calls. Particular actions like setting a filter or a buffer in the driver are handled through IOCTL functions.

3.8 Data Structures and Functions

3.8.1 Data structures

In this section we give a simple description of the most important data structures of the driver. The structures will not be described in deep because our goal is to give an overview of them and to point out the location of the most important driver's variables.

The main data structures of the packet capture driver are:

System structures

The following is a list of the kernel and NDIS data structures mostly used by the driver. We provide only a short description of them and of their use in the driver.

NDIS_PACKET This NDIS structure defines the packet descriptors with chained buffer descriptors for which pointers are passed to many `NdisXxx`, `MiniportXxx`, and

ProtocolXxx functions. A protocol driver that wants to write a packet to the network through the NDIS primitives must encapsulate it in a NDIS_PACKET structure. Similarly, a protocol driver receives a packet from the underlying NIC drivers in a NDIS_PACKET structure. Therefore, this structure is used heavily by the *PacketRead* and *PacketWrite* functions.

NDIS_PROTOCOL_CHARACTERISTICS This NDIS structure is used by the *DriverEntry* function during the initialization of the packet capture driver and contains all the information that NDIS will need to interact with the driver: the name of the protocol, the requested version of NDIS, the pointer to the various callback functions, and so on. This data is passed to NDIS through the *NdisRegisterProtocol* function.

LIST_ENTRY This is a kernel structure defined both in Windows 9x and in Windows NTx, to support handling of doubly linked lists. The kernel provides a set of primitives to handle lists, insert and remove elements, and so on. To use these primitives the LIST_ENTRY data structure must be used. All the linked lists of the drivers are handled through this structure.

IRP This structure is used only by the NT/2000 versions of the driver. IRP, or I/O Request Packet, is the fundamental structure that the Windows NTx kernels use for the communication between applications and drivers. All the data received by the driver from an application are packed in an IRP. All the data that the driver passes to the applications must be packed in an IRP. IRP is a VERY complex structure that contains all the data needed to communicate with the user-level: addresses of buffers, process IDs...

OPEN_INSTANCE This structure is used by almost all the functions of the driver. It contains the variables and the data associated with a running instance of the driver. The OPEN_INSTANCE structure is enough to identify completely and univocally an instance of the driver. A new instance of the driver with its own OPEN_INSTANCE structure is associated to every application that performs *CreateFile()* system call on the packet

capture driver. This means that more than one application can have access to the packet capture driver.

INTERNAL_REQUEST This structure is used by the driver to perform OID query and set operations on the network adapter through the underlying NIC driver. The query/set operations on the adapter can be done usually only by protocol drivers, but the packet capture driver allows the user-level applications to use this mechanism through an IOCTL function. The driver uses the **INTERNAL_REQUEST** structure to store the information on a query/set operation requested by the application.

DEVICE_EXTENSION This structure contains information on the packet capture driver, like the pointer to the **DRIVER_OBJECT** structure that describes the driver, or the *ProtocolHandle* that NDIS associates to the driver. It is used by the *DriverEntry* and *PacketUnload* functions to initialize or uninstall the driver

timeval This structure is used to store the timestamp associated with a packet

3.8.2 Functions

This paragraph will describe the procedures of the packet capture driver.

DriverEntry

This procedure is called by the operating system when the packet capture driver is loaded and started. It makes all the initializations needed by the driver. In particular this function allocates a **NDIS_PROTOCOL_CHARACTERISTICS** structure and initializes it with the protocol data (version, name, etc.) and the addresses of all the callback functions of the driver. This structure is then passed to NDIS with a call to *NdisRegisterProtocol*. The Windows NT version of the driver calls also the *IoCreateDevice* function to pass to the operating system the addresses of the handlers for the open/close, read, write and IOCTL requests.

PacketOpen

This function is called when a new instance of the driver is opened. It allocates and initializes variables and buffers needed by the new instance, fills the OPEN_INSTANCE structure associated with it and opens the adapter. The time constants used to obtain the timestamps of the packets during the capture are initialized here.

PacketOpenAdapterComplete

Callback function associated with the *NdisOpenAdapter* function of NDIS library. It is invoked by NDIS when the NIC driver has finished an open operation that was started previously by the *PacketOpen* function with a call to *NdisOpenAdapter*.

PacketClose

This function is called when a running instance of the driver is closed. It stops the capture and buffering process and deallocates the memory buffers that were allocated by the *PacketOpen* function. The network adapter is closed here with a call to *NdisCloseAdapter*.

PacketReset

Resets the adapter associated with the current instance, calling the *NdisReset* function of NDIS library. This function is defined only in Windows 95, because the Windows NT version of the driver calls *NdisReset* directly from the *PacketIoControl* function.

PacketUnload

This function is called by the system when the packet capture driver is unloaded. It frees the DEVICE_EXTENSION internal structure, and calls *NdisDeregisterProtocol* to deregister from NDIS.



9356
TH-HL

PacketIoControl

Once the packet capture driver is opened it has to be configured from user-level applications with IOCTL commands using the *DeviceIoControl* system call. This function handles IOCTL calls from the user level applications.

PacketWrite

This function is used to send packets to the network. The packet to send is passed to the driver with the *WriteFile* system call.

PacketSendComplete

Callback function associated with the *NdisSend* function of NDIS library. It is invoked by NDIS when the NIC driver has finished to send a packet to the network, after *PacketWrite* was called. This function frees the NDIS_PACKET structure that was allocated in the *PacketWrite* function, and awakens the application from the *WriteFile* or *DeviceIoControl* system call.

PacketRead

This function is invoked when the user level application performs a *ReadFile* system call (in Windows NTx), or an IOCTL_PROTOCOL_WRITE IOCTL call (in Windows 95x). In both cases the application must provide a buffer that will be filled by the driver with the packets coming from the network. The behavior of this function is the same when the driver is in mode 0 and when it is in mode 1.

The first operation performed by *PacketRead* is a check on the driver's circular packet buffer associated with the current instance of the driver. There are two possible cases:

1. The packet buffer is empty. This is ALWAYS true if current instance of the driver is in mode 1. The application's request cannot be satisfied immediately, and the system call

must be blocked until at least a packet arrives from the net or the timeout on this read expires. *PacketRead* sets the timeout on this read calling *KeSetTimer* in Windows NTx, and the custom function *SetReadTimeOut* in Windows 9x. After a *NDIS_PACKET* structure is allocated, the buffer received from the application is mapped in the driver's address space and associated with this structure. The *NDIS_PACKET* structure is put in the linked list of pending reads. It will be extracted from this list by the *Packet_Tap* function when a new packet will be captured. At this point *PacketRead* returns without awaking the application.

2. The packet buffer contains data: the application's request can be satisfied immediately. First, the driver obtains the length of the buffer passed by the application, to which the packets will be copied. Knowing this value, *PacketRead* tries to copy all the data to the application without further operations. This is possible if the number of bytes present in the driver's circular buffer is smaller than the size of the application buffer. If the application's buffer is too small to contain all the data present in the driver's packet buffer, a scan on the kernel buffer is performed to determine the amount of bytes to copy. After the scan, *NdisMoveMemory* is invoked to copy the packets. At this point the application is awaked and *PacketRead* returns.

PacketMoveMem

This function is used by all versions of the driver to copy data from the driver's circular buffer to the user-level application's buffer. It copies the data minimizing the accesses to the RAM. This is obtained aligning the memory accesses at the dword. Furthermore, it updates the head of the circular buffer every 1024 bytes copied. In this way the circular buffer is updated during the copy allowing a better use of the circular buffer and a lower loss probability. The function has been written to have a low overhead compared to a normal copy function. For this reason the head is updated no more often than every 1024 bytes.

ReadTimeout

This function is automatically invoked by the kernel when the timeout associated with a read call expires. First of all, the `OPEN_INSTANCE` structure associated with the current instance of the driver is found. From this structure `ReadTimeout` determines the working mode of current driver's instance. There are two possibilities.

Current driver's instance is in mode 0. No packet passed the filter during the timeout period. In fact, if a single packet had passed the filter, the timeout would have been deleted in the `packet_tap` function. For the same reason, the driver's circular buffer must be empty. The read system call associated with current instance is completed, but an empty buffer is passed to the application.

Current driver's instance is in mode 1. `ReadTimeout` builds in the user-level buffer a `bpf_hdr` structure, and fills it with the current timestamp and with the value 16 for `bh_caplen` and `bh_datalen`. After the header, two 64 bit integers are put. The first is the count of the number of packets satisfying the filter (hold in the `Npackets` field of the `OPEN_INSTANCE` structure associated with current instance), while the second is the amount of bytes satisfying the filter (hold in the `Nbytes` field of the `OPEN_INSTANCE` structure). Finally, `Npackets` and `Nbytes` are reset and the read system call associated with current instance is completed.

PacketCancelRoutine

This is the cancel routine set in the `PacketRead` function with a call to `IoSetCancelRoutine`. It is called by the operating system when the read system call is cancelled, for example when the user-level application is closed during a read on the packet capture driver, `PacketCancelRoutine` removes the pending IRPs from the queue and completes them. This function was introduced to correct a bug of the old versions of the driver. Without this function, in fact, the driver hangs after the user-level application is closed until at least a packet arrives from the network. This is a Windows NT specific problem, so this function is present only in the Windows NT version of the driver.

Packet_tap

Packet_tap is invoked by the underlying NIC driver when a packet arrives to the network adapter. In Windows 95 and in Windows NT it has the same following syntax:

```
NDIS_STATUS Packet_tap ( NDIS_HANDLE ProtocolBindingContext,  
NDIS_HANDLE MacReceiveContext,  
PVOID HeaderBuffer,  
UINT HeaderBufferSize,  
PVOID LookAheadBuffer,  
UINT LookaheadBufferSize)
```

First of all, the *ProtocolBindingContext* parameter is used to determine if current instance is running in mode 0 or in mode 1.

Then, *Packet_Tap* executes the BPF filter on the packet. The filter is obtained from the OPEN_INSTANCE structure pointed by the *ProtocolBindingContext* input parameter. To optimize the capture performances and minimize the number of bytes copied by the system, the BPF filter is applied to a packet before copying it, i.e. when it is still in the NIC driver's memory.

Notice that the capture driver receives the incoming packet from NDIS in two buffers: one containing the header and one containing the data. The reason of this subdivision is that normally a protocol driver makes separate uses of the header (used to decode the packet), and the data (sent to the applications). This is not the case of the packet capture driver, that works at link-layer level and needs to treat the whole packet as a unit. However, we noted that in the great part of the cases the packet is stored in the NIC driver's memory in a single buffer (this is the most obvious choice, because the packet arrives to the NIC driver through a single transfer). In these situations *HeaderBuffer* and *LookAheadBuffer* point to two different sections of the same memory buffer, and it is possible to use *HeaderBuffer* as a pointer to the whole packet and use the standard *bpf_filter* function. The packet driver in every case performs a check on the

distance between the two buffers: if it is equal to *HeaderBufferSize* (i.e. there is a single buffer), the standard *bpf_filter* function is called, otherwise *bpf_filter_with_2_buffers* is called.

If the filter accepts the packet, there are two possibilities:

Current instance is working in mode 1. In this case, *packet_tap* is extremely fast, because it has to do very few operations. The *Npackets* field of the *OPEN_INSTANCE* structure associated with current driver's instance is incremented. The length of the packet (plus a correction that adds the length of the preamble, the SFD and the FCS) is added to the *Nbytes* field of the *OPEN_INSTANCE* structure. After this, *packet_tap* returns.

Current instance is working in mode 0, *Packet_Tap* tries to extract an element from the linked list of pending reads. This operation can have two results:

The list of pending reads is empty. This means that the user-level application is not waiting for a packet in this moment. The incoming packet must be copied to the circular buffer of the packet capture driver. The address of the circular buffer and the pointers to head and tail are obtained from the *OPEN_INSTANCE* structure pointed by the *ProtocolBindingContext* input parameter. If the buffer is full (or, better, if the incoming packet does not fit in the remaining buffer space), the incoming packet is discarded. *Packet_Tap* allocates a *NDIS_PACKET* structure to receive the packet, and associates to this structure the correct memory position in the circular buffer. At this point, the amount of bytes specified previously by the filter is copied from the NIC driver's memory, and the *bpf_hdr* structure for this packet is built. Then the head and tail of the buffer are updated and *Packet_Tap* returns.

The list of pending reads contains at least one element. This means that at the moment the user-level application is blocked waiting for the result of a read on the packet driver. *Packet_Tap* extracts the first pending read from the list and finds the *NDIS_PACKET* structure associated with it. This structure is used to receive the packet from the NIC driver. At this point, the amount of bytes specified previously by the filter

is copied from the NIC driver's memory, and the *bpf_hdr* structure for this packet is built. Finally the application is awaked and *Packet_Tap* returns.

To build the *bpf_hdr* structure associated with a packet, current value of the microsecond timer must be obtained from the system. In Windows NT this is done by means of a call to the kernel function *KeQueryPerformanceCounter*, in Windows 95 with a call to the packet driver's function *QuerySystemTime*. Since *Packet_Tap* is called directly by the NIC driver, the receive timestamp is closer to the actual reception time.

PacketTransferDataComplete

This function is called by *Packet_Tap* when the packet must be passed directly to the user-level application. *PacketTransferDataComplete* releases the *NDIS_PACKET* structure and the buffers associated with the packet and awakes the application.

PacketRequest

This function is used to send a *OID* query/set request to the underlying NIC driver. *PacketRequest* allocates a *INTERNAL_REQUEST* structure and fills it with the data received from the application. Then The *NdisRequest* function from *NDIS* library is used to send the request to the NIC driver. This function is available only in Windows 95/98, because in Windows NT/2000 this operation is performed by the *PacketIoControl* function.

PacketRequestComplete

Callback function associated with the *NdisRequest* function of *NDIS* library. It is invoked by *NDIS* when the NIC driver has finished an open operation that was started previously either by the *PacketRequest* or *PacketIoControl* function of the packet capture driver.

GetDate

This assembler function returns the current system date as a 64 bit integer. The low word contains the current time in milliseconds. The high word contains the number of days since January 1, 1980. This function is present only in the Windows 95 version of the driver.

QuerySystemTime

This assembler functions returns the current microsecond system time as a 64 bit integer. This function is present only in the Windows 95 version of the driver, where it is used to get the timestamps of the packets.

CHAPTER 4

PACKET MONITORING

Packet monitoring implies capturing all packets that travel in the LAN. This can be achieved in an Ethernet network because of its broadcasting nature. That is any packet sent over an Ethernet network is broadcasted to all machines in the network. The Ethernet card in every machine checks whether the particular packet is destined to it, if so accepts it else rejects it. This basic functionality of an Ethernet Network is exploited to capture all packets that travel through the network. The Ethernet card can be placed in a number of modes using drivers and can be used to capture packets as desired.

4.1 Modes in an Ethernet Interface Card

The various modes in which an Ethernet card can be placed is as follows:

- Broadcast.
- Multicast.
- Directed.
- Promiscuous.

Broadcast: A frame that is to be sent to all machines in the network is broadcasted over the network with the destination address 0xfffff. This is supposed to be the broadcast address for the frame. Any card that is placed in the broadcast mode accepts frames with this destination address. Usually all cards will be configured to accept broadcast frames.

Multicast: A frame that is to sent to a particular set of machines is sent as a multicast frame with a specified multicast address as its destination address. These sets of machines constitute a multicast group. Thus any machine that is a member of the multicast group will accept the frame with the multicast destination address. Even though a card is not a part of the multicast group it can programmatically be placed in the multicast mode to accept all multicast frames.

Directed: A frame that is destined to a particular machine will have the destination machine's physical address (Ethernet address) specified as its destination address. The machine with that physical address will accept the frame while all the others will reject it. The card can also be set to only receive directed frames programmatically.

Promiscuous: Any card when placed in this mode accepts any packet that comes to it. This mode along with the broadcasting nature of the Ethernet is the key to the packet monitoring application.

The PACKET.SYS sample driver that comes along with windows NT DDK helps to place the network card in all the above mentioned modes. The application with the aid of the PACKET.SYS places the card in the promiscuous mode to capture all the packets that travel in the network.

4.2 Sample PACKET.SYS Driver

This driver helps to place the network card in any mode desired and also allows applications to send and receive packets through and from the network. The driver sample apart from the sys file also provides a DLL (PACKET32.DLL) through which an application communicates with the driver.

The diagram shown below clearly depicts the way our application interacts with the PACKET.SYS driver. The Application calls the functions in the DLL, which in turn

calls the entry points in the PACKET.SYS driver. The driver uses the NDIS.SYS exported functions to communicate with the Network Interface Card.

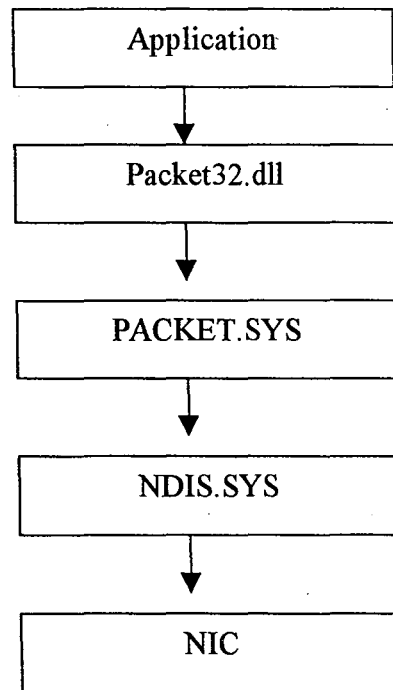


Figure: 4 Communication between application and PACKET.SYS

The structures that can be used in the above process are listed below.

```
typedef struct _ADAPTER
{
    // holds the handle that is returned by the CreateFile method.
    HANDLE hFile;

    // Holds the symbolic link name of the driver.
    TCHAR SymbolicLink[MAX_LINK_NAME_LENGTH];
} ADAPTER, *LPADAPTER;

typedef struct _PACKET
{
    // Holds the handle to the event that is associated with the adapter object.
```

```

    HANDLE    hEvent;
    // The OVERLAPPED structure contains information used in
    // asynchronous input and output
    OVERLAPPED OverLapped;
    // Buffer that holds the data sent or received.
    PVOID     Buffer; //Length of the buffer
    UINT     Length;
} PACKET, *LPPACKET;
typedef struct _CONTROL_BLOCK
{ //pointer to the Adapter object
LPADAPTER hFile;
HANDLE    hEvent; //handle to the event
TCHAR    AdapterName[64]; //Name of the driver as registered in the registry
//Buffer where the received packet data is stored
    HANDLE    hMem;
    LPBYTE    lpMem;
    //Buffer where the packet data to be sent is stored
    HGLOBAL   hMem2;
    LPBYTE    lpMem2;
    //Specifies the packet length
    ULONG     PacketLength;
    //Specifies the last read packet size
    ULONG     LastReadSize;
    //Specifies the Buffer length
    UINT     BufferSize;
} CONTROL_BLOCK, *PCONTROL_BLOCK;

```

The various function definitions as provided in the Packet32.dll that are used by the application are as follows:

```

LONG PacketGetAdapterNames(PTSTR pStr, PULONG BufferSize)
{
    HKEY    SystemKey;
    HKEY    ControlSetKey;
    HKEY    ServicesKey;
    HKEY    NdisPerfKey;
    HKEY    LinkageKey;
    LONG    Status;
    DWORD   RegType;
    // Open the Key HKEY_LOCAL_MACHINE,
    Status=RegOpenKeyEx(HKEY_LOCAL_MACHINE,TEXT("SYSTEM"), 0,
        KEY_READ,&SystemKey);
    if (Status == ERROR_SUCCESS)
    {
        // Open the key currentcontrolset
        Status=RegOpenKeyEx(SystemKey,TEXT("CurrentControlSet"), 0,
            KEY_READ,&ControlSetKey);
        if (Status == ERROR_SUCCESS) { // Open the key Services
            Status=RegOpenKeyEx(ControlSetKey,TEXT("Services"),0,
                KEY_READ,&ServicesKey);
            if (Status == ERROR_SUCCESS) { // Open the key Packet.
                Status=RegOpenKeyEx(ServicesKey,TEXT("Packet"),0,
                    KEY_READ,&NdisPerfKey);
                if (Status == ERROR_SUCCESS) { // Open the key Linkage.
                    Status=RegOpenKeyEx(NdisPerfKey,TEXT("Linkage"), 0,
                        KEY_READ, &LinkageKey );
                    if (Status == ERROR_SUCCESS) { // Open the key Export.
                        Status=RegQueryValueEx(LinkageKey,TEXT("Export"),NULL,
                            &RegType,(LPBYTE)pStr,BufferSize );
                    }
                }
            }
        }
    }
}

```



```

    // Close all the keys that have been opened so far.
    RegCloseKey(LinkageKey);
}
RegCloseKey(NdisPerfKey);
}
RegCloseKey(ServicesKey);
}
RegCloseKey(ControlSetKey);
}
RegCloseKey(SystemKey);
}
return Status;
}

```

The above function PacketGetAdapterNames retrieves the name of the driver as registered in the registry.

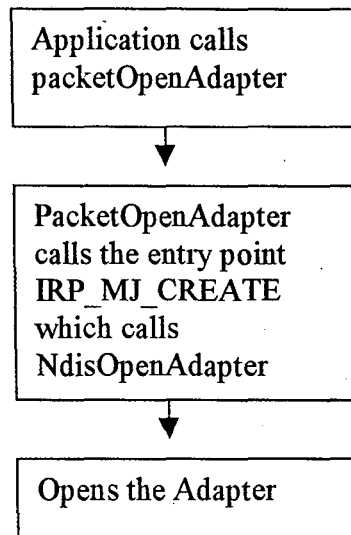


Figure : 5 Application interacting with the driver

The above diagram shows flow of a function call from the application. This applies to all other functions used to interact with the driver. The function PacketOpenAdapter defines a new DOS device name for the device and calls the

CreateFile method to create or open a communication device and get a handle to the device. This function is called from the application prior to sending or receiving packets. The CreateFile method invokes the entry point specified by the IRP_MJ_CREATE in the driver, which calls the NDIS library, exported function NdisOpenAdapter to open the Adapter.

```

PVOID PacketAllocatePacket(LPADAPTER AdapterObject )
{
LPPACKET lpPacket;
// Allocates memory for the Packet object.
lpPacket=(LPPACKET)GlobalAllocPtr(GMEM_MOVEABLE | GMEM_ZEROINIT,
sizeof(PACKET) );
if (lpPacket==NULL) {
ODS("Packet32: PacketAllocateSendPacket: GlobalAlloc Failed\n");
return NULL;
}
// Create an event that will be signaled when the operation is over.
lpPacket->OverLapped.hEvent=CreateEvent(
NULL,
FALSE,
FALSE,
NULL
);
if (lpPacket->OverLapped.hEvent==NULL) {
ODS("Packet32: PacketAllocateSendPacket: CreateEvent Failed\n");
GlobalFreePtr(lpPacket);
return NULL;
}
return lpPacket;
}

```

The above function `PacketAllocatePacket` allocates memory for the packet object and calls the `CreateEvent` function to create an event that is associated with the particular file handle.

```
VOID PacketInitPacket(LPPACKET lpPacket, PVOID Buffer,
    UINT Length )
{
    // Set packet object's buffer to the buffer.
    lpPacket->Buffer=Buffer;
    // Set packet object's buffer Length to the buffer Length.
    lpPacket->Length=Length;
}
```

This function sets the packet object's buffer to the buffer pointer that is passed.

```
BOOLEAN PacketReceivePacket(LPADAPTER AdapterObject,
    LPPACKET lpPacket,BOOLEAN Sync,PULONG BytesReceived )
{
    BOOLEAN Result;
    // Set offset value to 0.
    lpPacket->OverLapped.Offset=0;
    lpPacket->OverLapped.OffsetHigh=0;
    if (!ResetEvent(lpPacket->OverLapped.hEvent)) {
        return FALSE;
    }
    // Call ReadFile to read a packet.
    Result=ReadFile(
        AdapterObject->hFile,
        lpPacket->Buffer,
        lpPacket->Length,
        BytesReceived,
```

```

        &lpPacket->OverLapped
    );
    if (Sync) {
        Result=GetOverlappedResult(
            AdapterObject->hFile,
            &lpPacket->OverLapped,
            BytesReceived,
            TRUE
        );
    }
    else
    {
        Result = TRUE;
    }
    return Result;
}

```

The function calls the driver's appropriate entry point to read a packet from the network and place it in the buffer specified. This is accomplished by calling the ReadFile method with the handle returned by the CreateFile method.

These are the various routines that are useful towards packet monitoring using the driver PACKET.SYS.

4.3 TCP/IP layered architecture

In the TCP/IP each layer has a specific role.

Application Layer Network applications depend upon the definition of a clear dialog. In a client-server system, the client application knows how to request services, and the

server knows how to appropriately respond. Protocols that implement this layer include HTTP, FTP, and Telnet.

Transport Layer The Transport Layer allows network applications to obtain messages over clearly defined channels and with specific characteristics. The two protocols within

4	Application layer
3	Transport layer
2	Network layer
1	Link layer

Figure: 6 TCP/IP layered Architecture

the TCP/IP suite that generally implement this layer are *Transmission Control Protocol* (TCP) and *User Datagram Protocol* (UDP).

Network Layer :The Network Layer allows information to be transmitted to any machine on the contiguous TCP/IP network, regardless of the different physical networks that intervene. *Internet Protocol* (IP) is the mechanism for transmitting data within this layer.

Link Layer : The Link Layer consists of the low-level protocols used to transmit data to machines on the same physical network. Protocols that aren't part of the TCP/IP suite, such as Ethernet, Token Ring, FDDI, and ATM, implement this layer.

Data within these layers is usually encapsulated with a common mechanism: protocols have a *header*, identifying meta-information such as the source, destination, and other attributes, and a data portion that contains the actual information. The protocols from the upper layers are encapsulated within the data portion of the lower ones. When traveling back up the protocol stack, the information is reconstructed as it is delivered to each layer

4.4 Identifying Users Different Requests

The above section describes how all the packets in an Ethernet network can be captured. Our idea is not only to capture all packets but also to monitor the Internet

activity in the network, which means that we have also to identify packets carrying HTTP, FTP, SMTP, etc., requests. This requires knowledge about the Ethernet frame structure and how IP, TCP/UDP packets are encapsulated in a frame. This section describes how packets are identified as TCP/IP packets, how HTTP requests are identified and how URL information in a frame is obtained.

Data Flow in a TCP/IP Network

To transmit data across a layered network we pass data from our application to a protocol on a protocol stack. After that protocol finishes with our data it passes the data to the next protocol on the stack. As the data passes through each layer, the protocols on the stack encapsulates the data for the next lower level in the stack. Encapsulation, therefore, is the process of storing your data in the format required by the lower level protocol in the stack.

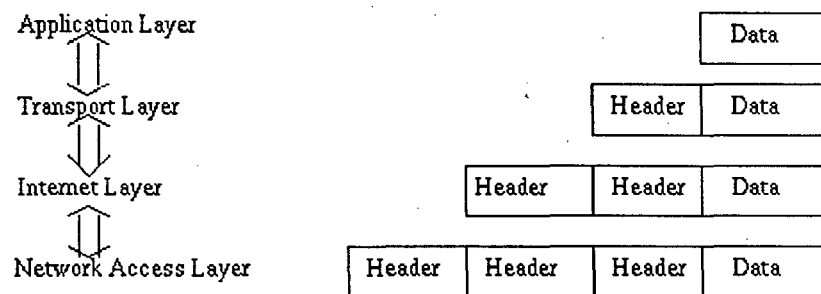


Figure: 7 Data flow in TCP/IP layered model

So we see that the application module encapsulate data from the user in an application message. TCP module encapsulates the application data and attaches the TCP header and sends it to the next layer. As the data passes through IP module in the network layer, it formats the TCP segment into an IP datagram or packet. The Ethernet driver formats the data from the IP module and places the data into an Ethernet frame. This explains how a frame encapsulates an IP datagram and further how the IP packet encapsulates TCP/UDP data.

4.4.1 Identifying TCP/IP packet

To identify a packet as a TCP/IP packet we have to first have a look at the Ethernet frame structure.

Number of bits	function
56	Preamble
8	Start Frame Delimiter
48	Destination Address
48	Source Address
16	Frame Type
0-12000	Data (LLC Data Bytes)
0-368	Padding
32	FCS

Figure: 8 Format of an Ethernet Data Frame.

The fields that are of our interest in the Ethernet header are:

- Destination address (6 Bytes).
- Source address (6 Bytes).
- Frame type (2 Bytes).

As the name suggests the destination address field specifies the destination of the Ethernet frame. Similarly the source address field specifies the source address of the frame. The frame type field is the field of our interest. This field identifies the overlying protocol of the frame.

If the packet is a valid packet then the value of Frame type field (13th and 14th bytes) will be 08 00 Hex.

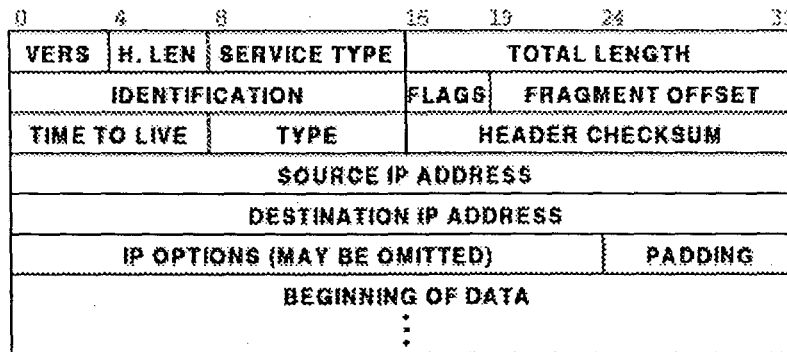


Figure: 9 IP Header Format

After identifying a packet as a TCP/IP packet, our next duty is to find out whether the packet is a TCP packet or any other packet. Since any HTTP request goes as only TCP request we can ignore other packets. To find out whether the packet is a TCP packet we have to parse the IP header. The IP header is shown above.

The important fields in the IP header are

- Header Length (4 Bits) and Version number (4 Bits).
- Total Packet Length (2 Bytes).
- Protocol to be used as the data passes upwards to the transport layer. The following table specifies the protocol field values for the common TCP/IP protocols that use IP.

Protocol	Value(decimal)
TCP	6
UDP	17
ICMP	1
IGMP	2

Figure: 10 Protocol field values in TCP/IP suite

- Header Checksum (2 Bytes).
- Source IP (4 Bytes).
- Destination IP (4 Bytes).

Thus a value of 06 in the protocol field of the IP header specifies that the packet is TCP packet.

4.4.2 Identifying a HTTP request

After identifying a packet as a TCP packet, we have to find out whether the packet is an HTTP request packet. To find out whether a packet contains an HTTP request we have to examine the TCP header. The TCP header is shown below.

The important fields in the TCP header are

- Source Port (2 Bytes).

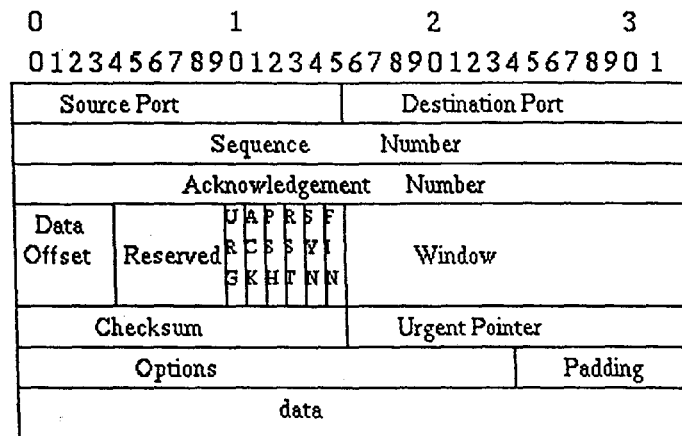


Figure: 11 TCP header format

- Destination Port (2 Bytes).
- Sequence Number (4 Bytes).
- Acknowledge Number (4 Bytes).
- Hlen, Reserved and Code Bits (2 Bytes).

- Window (2 Bytes).
- Checksum (2 Bytes).
- Urgent Pointer (2 Bytes).

Of the above mentioned fields in the TCP header, the fields of our interest are the source and the destination port fields. These fields specify the port to which a connection is established. Various services of the TCP like the HTTP, FTP, etc use specific port numbers to render their services.

For an HTTP service the port number would be $00\ 80_{10}$ or $00\ 50_{16}$.

If the source port field contains the value of 80 (HTTP port number) then the packet is an HTTP response packet. If the destination port field is 80 then the packet is HTTP request packet.

Once the HTTP request packet is found it is fairly easy to get the URL contained in the data part of the packet. Any request we type in a browser goes to the HTTP server as GET or a POST request. The browser attaches other browser-related information and sends the packet to the corresponding HTTP server. The whole information including the URL of the page requested is encapsulated with in the data part of the packet. Thus just by parsing the data part of the packet we can obtain the URL.

CHAPTER 5

NETWORK BANDWIDTH CONTROL

In the previous chapter we examined how the packets can be captured by placing the Ethernet network card in the promiscuous mode. After capturing the packets we can extract the URL contained in them. With these URLs that are obtained it is possible to find out the site the particular user has visited. We will also see how to control network bandwidth when the load on the network exceeds the value that we set up in our policies.

5.1 TCP communication in a network :

To understand this we see the various fields that are present in the TCP header

0		16		31
SOURCE PORT (2 bytes)		DESTINATION PORT (2 bytes)		
SEQUENCE NUMBER (4 bytes)				
ACKNOWLEDGEMENT NUMBER (4 bytes)				
HLEN (4 bits)	RESERVED (6 bits)	CODE BITS (6 bits)	WINDOW (2 bytes)	
CHECK SUM (2 bytes)			URGENT POINTER(2bytes)	
OPTIONS (if any 3 bytes)			PADDING (1 byte)	
DATA...				

Figure: 12 TCP header format

The important fields in the TCP header are

- ☞ Source Port (2 Bytes).
- ☞ Destination Port (2 Bytes).
- ☞ Sequence Number (4 Bytes).

☞ Sequence Number (4 Bytes).

The various fields that are of interest to us are elaborated below:

Source Port: This field specifies the port from which the request originated. The source port value can be used to find out whether the packet is a HTTP packet, FTP packet etc.

Destination Port: This field specifies the port to which the packet is destined. The destination port value can be used to find out whether the packet is a HTTP response, FTP response etc.

Sequence Number: The sequence number identifies the position of the packet in the sender's byte stream.

Acknowledgement Number: The acknowledgement number field identifies the number of the octet that the source expects to receive next.

Header Length: Since TCP header includes the variable length options and padding fields, the length of the TCP header is not fixed. This field specifies the length of the header. This is a 4-bit field.

Reserved Bits: This is a 6-Bit field and is reserved for future use.

Code Bits: The code bits totally comprise 6 bits and each has its own significance.

The six code bits are:

URG: This specifies that the packet has some urgent data in it. The urgent pointer field that follows will hold the offset from where the urgent data begins.

ACK: This signifies that the acknowledgement number field is valid.

PSH: This specifies that this segment requests a push. This flag is set usually for packets that contain TCP requests or responses.

RST: This is the reset bit that tells the host to reset the connection. That is, if this bit is set then it means that the connection has to be reset immediately.

SYN: This bit specifies that the sequence number is valid. In other words this bit tells both the machines to synchronize the sequence numbers. Usually set when the hosts try for a connection.

FIN: This specifies that the sender has reached the end of the byte stream.

Checksum: This field contains a 16-bit integer checksum that is used to verify the integrity of the data as well as the TCP header.

Now that we have had a look at the TCP header, we can see how a connection is established between a host and a server.

5.2 Establishing a TCP connection

To establish a connection, TCP uses a three-way handshake. First the host that wants to connect sends a request with the SYN bit set to the server then the server replies with another segment in which both the SYN and the ACK bits are set. Then the host again sends a third segment that has the ACK bit alone set. With this the three-way handshake gets over and the connection is established between the two machines. Once the connection is established the data transfer can occur both ways.

5.3 TCP connection Reset

Apart from the normal closing operation, there may be certain abnormal conditions that force the application or the network software to break a connection. TCP provides the reset facility for such conditions.

To reset a connection, one side initiates the reset by sending a segment that has the RST bit set. The other side responds immediately by aborting the connection. This reset is instantaneous, in the sense that transfer in both directions ceases immediately and all the resources such as buffers are released.

5.4 Breaking a TCP connection (blocking a particular workstation)

As we have seen to break an ongoing TCP communication it is enough if we send a segment that has the RST bit set to anyone side of the connection. Once a machine receives this segment it immediately breaks the connection. The following illustration will make this concept clear.

Consider that a machine M1 in our LAN is accessing a server S1 through the Internet. The machine M1 will send a TCP request packet to the server S1, which will in turn send a response packet to M1 and their communication is established.

To break a connection we have to send a segment with the RST bit set to either machine M1 or to the server S1. Consider the situation where the machine M1 has sent a request to the server S1 and is waiting for the response. The request packet that is sent to the server S1 is captured by our monitoring application that is capable of capturing all the packets that travel in the network. Once we get this packet we immediately send the packet that has the RST bit set to the machine M1, since LAN is much faster than the Internet our packet will reach machine M1 before the response from the actual server. Assuming for now that the packet that we send is properly framed, it will be accepted by the IP and the TCP software on machine M1. Thus machine M1 will think that the server S1 has initiated a reset and will immediately reset the connection. The actual response that comes from the original server is ignored. The above section describes the logic involved in blocking a workstation from accessing the Internet. But, the catch lies in the framing of the packet with the RST bit set and sending it to the machine M1.

5.5 Framing of a RST packet

The packet that we send to machine M1 has to be framed properly so that the IP and the TCP software in the target machine accept it.

The various data that are required for framing a reset packet are:

- Source Ethernet Address.
- Destination Ethernet Address.
- Source IP Address.
- Destination IP Address.
- Total length of the packet as given by the IP header length field.
- IP Checksum.
- Source port.
- Destination port.

- Sequence number.
- Acknowledgement number.
- TCP Checksum.

The values for most of the above fields are picked up from the packet that we obtain. Let us see how the values in these fields are filled. Let us call the packet that we have captured as P1 and the new packet that we are framing as P2.

The **source Ethernet address** of P2 is the **destination Ethernet address** in P1.

The **destination Ethernet address** of P2 is the **source Ethernet address** in P1.

The **source IP address** of P2 is the **destination IP address** in P1.

The **destination IP address** of P2 is the **source IP address** in P1.

The total length of the packet is the length of the packet in bytes minus the Ethernet header length.

The IP checksum is then computed using the standard algorithm for the IP checksum computation routine that has been listed in the RFC.

The **source port** of P2 is the **destination port** in P1.

The **destination port** of P2 is the **source port** in P1.

The sequence number and the acknowledgement numbers are ignored so we need not fill anything in these fields.

The TCP checksum is computed using the standard algorithm for TCP checksum computation as listed in the RFC.

Here is the pseudo code of a routine called SendPacket that frames a reset packet and sends this packet to the target machine. Along with this the IP and the TCP checksum computation routines are also given.

```
CONTROL_BLOCK Adapter;
void SendPacket( int gnSend )
{
    short int nSendIndex;
    unsigned short *pIP,*pTcp;
    int nAckNum,finalAck;
    SeqAndAckNos ackNum;
```

```

Length lTotalLength;
Checksum ipChksum,tcpChksum;
unsigned char IPHead[20]; // IPAddr[8];
PVOID Packet;
char seqNo[20],ackNo[20],szAddr[25];
::FillMemory(seqNo,20,'\0');
::FillMemory(ackNo,20,'\0');
::FillMemory(szAddr,25,'\0');
// Allocate memory for the packet to be sent to the Ws.
Packet=PacketAllocatePacket((LPADAPTER)Adapter.hFile);
if (Packet != NULL)
{
PacketInitPacket( (PACKET *)Packet, Adapter.lpMem2, 54 );
/*****Start Filling the Packet to be sent *****/
/***** Ethernet header *****/
// 14 bytes
// Destination and Source ethernet address.
for(nSendIndex = 0;nSendIndex < 6;nSendIndex++)
{
// Copy the packets destination address
// into our source address and the source address
// into our destination address.
Adapter.lpMem2[nSendIndex] = pdBlockedData[gnSend].pData[nSendIndex+6];
Adapter.lpMem2[nSendIndex+6] = pdBlockedData[gnSend].pData[nSendIndex];
}
// Protocol type is Internet protocol.
Adapter.lpMem2[12] = 0x08;
Adapter.lpMem2[13] = 0x00;
/***** IP header details. *****/
// 20 bytes
// Version and Header length.

```



```

IPHead[0] = Adapter.lpMem2[14] = 0x45;
// Type of service. Normal.
IPHead[1] = Adapter.lpMem2[15] = 0x00;
// Total packet length
IPHead[2] = Adapter.lpMem2[16] = 0x00;
IPHead[3] = Adapter.lpMem2[17] = 0x28;
// Also get total packet length of the obtained packet.
ITotalLength.szTotalLength[1] = pdBlockedData[gnSend].pData[16];
ITotalLength.szTotalLength[0] = pdBlockedData[gnSend].pData[17];
finalAck = ITotalLength.nTotalLength;
// Identification number.
IPHead[4] = Adapter.lpMem2[18] = pdBlockedData[gnSend].pData[18];
IPHead[5] = Adapter.lpMem2[19] = pdBlockedData[gnSend].pData[19];
// flags + fragment offset.. set it to nothing.
IPHead[6] = Adapter.lpMem2[20] = 0x00;
IPHead[7] = Adapter.lpMem2[21] = 0x00;
// Time to live for the packet. no significance for selecting this 77.
IPHead[8] = Adapter.lpMem2[22] = 0x77;
// Protocol used (TCP)
IPHead[9] = Adapter.lpMem2[23] = 0x06;
// previous checksum.
IPHead[10] = 0x00;
IPHead[11] = 0x00;
// source IP address and destination IP address.
for(nSendIndex = 26;nSendIndex < 30;nSendIndex++)
{
// As with the ethernet address, do the same swapping with the IP
// addresses also.
IPHead[nSendIndex-14] = Adapter.lpMem2[nSendIndex] =
pdBlockedData[gnSend].pData[nSendIndex+4];
IPHead[nSendIndex-10] = Adapter.lpMem2[nSendIndex+4] =

```

```

pdBlockedData[gnSend].pData[nSendIndex];
}
// Set the pointer to point to the buffer where the IP header lies.
pIP = (unsigned short *)&IPHead[0]; // Call the checksum computation routine.
ipChecksum.iChecksum = htons( IpChecksum( pIP ) );
::ZeroMemory( szAddr, 25 );
// Fill the computed checksum
Adapter.lpMem2[24] = ipChecksum.szChecksum[0];
Adapter.lpMem2[25] = ipChecksum.szChecksum[1];
::ZeroMemory( szAddr, 25 );
/***** Fill the TCP header *****/
// 20 bytes.
// Source Port
Adapter.lpMem2[34] = pdBlockedData[gnSend].pData[36];
Adapter.lpMem2[35] = pdBlockedData[gnSend].pData[37];
// Destination port
Adapter.lpMem2[36] = pdBlockedData[gnSend].pData[34];
Adapter.lpMem2[37] = pdBlockedData[gnSend].pData[35];
// Sequence number
Adapter.lpMem2[38] = pdBlockedData[gnSend].pData[42];
Adapter.lpMem2[39] = pdBlockedData[gnSend].pData[43];
Adapter.lpMem2[40] = pdBlockedData[gnSend].pData[44];
Adapter.lpMem2[41] = pdBlockedData[gnSend].pData[45];
// Acknowledgement number computation
ackNum.szSeqOrAck[0] = pdBlockedData[gnSend].pData[38];
ackNum.szSeqOrAck[1] = pdBlockedData[gnSend].pData[39];
ackNum.szSeqOrAck[2] = pdBlockedData[gnSend].pData[40];
ackNum.szSeqOrAck[3] = pdBlockedData[gnSend].pData[41];
// compute ack number as the sum of seq number
// "plus"
// Number of bytes received

```

```

// "minus"
// 40 ( header length of IP and TCP )
// Convert every thing to network-Byte order
::ZeroMemory( szAddr, 25 );
// convert the ack number from
// Network-byte order to host byte order.
nAckNum = htonl(ackNum.nNumber);
::ZeroMemory( szAddr, 25 );
// increment the ack number by one alone.
nAckNum = finalAck - 40; // convert it back to host-byte order
ackNum.nNumber = htonl(nAckNum);
// fill the ack. number to be sent
Adapter.lpMem2[42] = ackNum.szSeqOrAck[0];
Adapter.lpMem2[43] = ackNum.szSeqOrAck[1];
Adapter.lpMem2[44] = ackNum.szSeqOrAck[2];
Adapter.lpMem2[45] = ackNum.szSeqOrAck[3];
// Header length + reserved + flags
Adapter.lpMem2[46] = 0x50;
Adapter.lpMem2[47] = 0x14;
// Window. For the reset packet 0 is enough.
Adapter.lpMem2[48] = 0x00;
Adapter.lpMem2[49] = 0x00;
// Checksum computation.
// Fill zero initially in the checksum field.
Adapter.lpMem2[50] = 0x00;
Adapter.lpMem2[51] = 0x00; // Urgent pointer
Adapter.lpMem2[52] = 0x00;
Adapter.lpMem2[53] = 0x00;
// set the corresponding pointers to be passed to the checksum
// computation routine.
pIP = (unsigned short *)&IPHead[12];

```

```

pTcp = (unsigned short *)&Adapter.lpMem2[34];
// Call the checksum computation routine.
tcpChecksum.iChecksum = TcpChecksum( pTcp, pIP );

// Fill the actual checksum value.
Adapter.lpMem2[50] = tcpChecksum.szChecksum[0];
Adapter.lpMem2[51] = tcpChecksum.szChecksum[1];
// Send the packet to the blocked workstation.
PacketSendPacket( (LPADAPTER)Adapter.hFile, (PACKET *)Packet, TRUE );
// Free the allocated pool.
PacketFreePacket((PACKET *)Packet);
}
}

```

The algorithm that computes the IP Check sum is given below:

The checksum field is the 16 bit one's complement of the one's complement sum of all 16-bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero.

```

unsigned short IpChecksum(unsigned short *pIP)
{
// variable holding the value of the IP header length.
unsigned short nLen=20;
// Variable that holds the checksum value.
unsigned chksum = 0;
// Variable that holds the value of the final checksum after
// computing the one's complement.
unsigned short finalchk;
// Divide the length by 2 for computing 16-bit checksum.
nLen >>= 1;

```

```

// compute the 16 bit checksum.
for(int i = 0; i < nLen;i++)
{
chksum += htons(*pIP++);
}
// Fold the 32 bit value into the 16-bit field.
chksum = (chksum >> 16) + (chksum & 0xffff);
chksum += (chksum >> 16);
// Compute the one's complement of the checksum.
finalchk = (~chksum & 0xffff);
// return the final checksum.
return finalchk;
}

```

The algorithm that computes is the TCP Check sum is given below:

The checksum field is the 16 bit one's complement of the one's complement sum of all 16-bit words in the header and text. If a segment contains an odd number of header and text octets to be checksummed, the last octet is padded on the right with zeros to form a 16-bit word for checksum purposes. The pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros. The checksum also covers a 96-bit pseudo header conceptually prefixed to the TCP header. This pseudo header contains the Source Address, the Destination Address, the Protocol, and TCP length. This gives the TCP protection against misrouted segments. This information is carried in the Internet Protocol and is transferred across the TCP/Network interface in the arguments or results of calls by the TCP on the IP.

The TCP Length is the TCP header length plus the data length in octets (this is not an explicitly transmitted quantity, but is computed), and it does not count the 12 octets of the pseudo header.

The routine for the above algorithm has been described below

```

unsigned short TcpChecksum(unsigned short *pTcp, unsigned short *pIP )
{
// Variable that holds the checksum not complemented.
unsigned chksum = 0;
// Variable that holds the final checksum value.
unsigned short finalchk;
// Variables that the TCP & IP header lengths.
unsigned short nLen = 20, nIpLen = 8;
// divide both length values by 2 since we are adding 16 bits.
nIpLen >>= 1;
nLen >>= 1;
// compute the 16 bit sum of the pseudo header.
for(int i = 0;i < nIpLen;i++)
{
chksum += (*pIP++);
}
// length + the protocol type.
chksum += htons( 26 );
// compute the 16 bit sum of the TCP header and data.
for(i = 0;i < nLen;i++)
{
chksum += (*pTcp++);
}
// Fold the 32 checksum into 16 bit.
chksum = (chksum >> 16) + (chksum & 0xffff);
chksum += (chksum >> 16);
// Get the one's complement of the checksum.
finalchk = (~chksum & 0xffff);
// return the final checksum.
return finalchk;
}

```

The above algorithm describes how a particular user of the network can be restricted from accessing the Internet by framing a RST packet and sending the packet to the user and how to compute TCP and IP checksums.

Conclusion

This work has been modest attempt to study the different techniques used for managing the bandwidth and to develop a software that manages the bandwidth of Jawaharlal Nehru University (JNU) campus wide network. The salient features of the approach used in the work are

- Easy installation in the network: Most WAN access links are attached to campus networks via a router on an Ethernet LAN. Installation of a system becomes installing one or more machines on the same LAN.
- Network unaffected by manager hardware failure: In the approach that we described here If the manager machine fails, the controls stop working, but the network traffic continues to flow in the original uncontrolled manner.
- Efficient utilization of the bandwidth capacity of the Internet access link.

The work can be extended to limit the number of Internet connections to a particular group of users.

An alternative approach to control the bandwidth of a network is TCP connection admission control. Connection admission control appears to guarantee some TCP throughput performance on an overloaded Internet access link. In this approach we can explore better algorithms to invoke connection admission control.

References

1. Andrew S. Tanenbaum, "Computer Networks," Third Edition, Prentice Hall of India, 1999.
2. Anurag Kumar and Malati Hegde, "Nonintrusive TCP Connection Admission Control for Bandwidth Management of an Internet Access Link," IEEE Communication Magazine, May 2000.
3. Douglas E. Comer, "Computer Networks and Internet," Second Edition, Prentice Hall of India, 2000.
4. Douglas E. Comer, "Internetworking With TCP/IP," Vol 1: Principles, Protocols, and Architecture, Third Edition, Prentice Hall of India, 1999.
5. Dr. J. Nevil Brownlee, "Internet Traffic Measurement: an Overview," A background paper for the ICAIS seminar Miyazki, Japan, 8 March 1999.
6. G. Viscarola & W. Anthony Mason, "Windows NT Device Driver Development," First Indian Edition, Techmedia, 1999.
7. Gray. R. Wright and W. Richard Stevens, "TCP/IP illustrated, Volume 2 The Implementation," First ISE 1999.
8. K. G. Coffman and Andrew Odlyzko, "The size and the growth rate of Internet," Study available at http://www.firstmonday.dk/issues/issue3_10/coffman/index.html.
9. Postel, J. B., ed. 1981a. "Internet Protocol," RFC 791, 45 pages (Sept.)

10. Postel J. B., ed. 1981c. "Transmission Control Protocol," RFC 793, 85 pages (Sept.)
11. R. Apparna and B. Prem Kumar, "Monitoring Ethernet Network activity with NDIS drivers," California Software Laboratories, January 1999.
12. R. Mandeville and D. Newman, "Traffic Tuners: Striking the Righth Note?," Data Commun. (Asia-Pacific), Nov. 21, 1998.
13. Stallings William, "Data and Computer Communications," Fifth Edition, Prentice Hall of India.
14. Scott Mace, "Breaking Bandwidth Bottlenecks," Byte Magazine, May 1998.
15. W. Richard Stevens, "TCP/IP Illustrated," Volume 1: The Protocols, Addison Wesley, 2000.