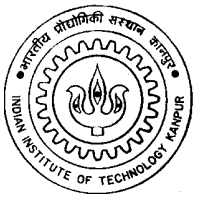
Gigabit PickPacket: A Network Monitoring Tool for Gigabit Networks

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by

P Satya Srikanth



to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

May, 2004

Certificate

This is to certify that the work contained in the thesis entitled "*Gigabit Pick-Packet: A Network Monitoring Tool for Gigabit Networks*", by *P Satya Srikanth*, has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

May, 2004

(Dr. Dheeraj Sanghi)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

(Dr. Deepak Gupta)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Abstract

The extensive use of computers and networks for exchange of information has also had ramifications on the growth and spread of crime through their use. Law enforcement agencies need to keep up with the emerging trends in these areas for crime detection and prevention. Among the several needs of such agencies are the need to monitor, detect and analyze undesirable network traffic. However, the monitoring, detecting, and analysis of this traffic may be against the goal of maintaining privacy of individuals whose network communications are being monitored. Also, the bandwidth at network backbones and Internet Service Providers is increasing rapidly due to the increase in network usage. This increase in bandwidth imposes an additional requirement on Network Monitoring Tools to monitor traffic at very high speeds without losing any relevant information.

PickPacket — a network monitoring tool that can handle the conflicting issues of network monitoring and privacy through its judicious use, is discussed in References [1, 8, 9, 12]. This thesis discusses the design and development of a network monitoring tool called Gigabit PickPacket, an enhanced version of PickPacket for monitoring network at Gigabit speed. This tool effectively uses the support of multiprocessor and/or multiple machines for monitoring traffic at very high speeds.

Acknowledgments

I take this opportunity to express my sincere gratitude towards my thesis supervisors Dr. Dheeraj Sanghi and Dr. Deepak Gupta for their invaluable guidance. It would have never been possible for me to take this project to completion without their innovative ideas and encouragement. I also thank the other team members involved with the development of PickPacket for their cooperation and support. Sanjay Jain helped me initially to understand the PickPacket architecture thoroughly. Ramesh helped me at every point in the development of Gigabit PickPacket. Sudheer and Ananth helped me by implementing a splitter and explained me the problems associated in building a splitter at software level. I would like to thank BrajeshJi, for being cooperative and allowing me to use the figures in his thesis report.

I also wish to thank whole heartily all the faculty members of the Department of Computer Science and Engineering, IIT Kanpur for enhancing my knowledge. I also wish to thank Navpreet SinghJi for helping me to sniff on the CC network. I would like to thank whole of the mtech2002 batch for the times I shared with them. My special thanks to Saradhi for helping me whenever I faced problems in using any tool in Linux. I would like to thank everyone in the Prabhu Goel Research Centre for providing a nice and challenging work environment. I would also like to thank the Prabhu Goel Research Centre for sponsoring my work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

There has been a tremendous growth in the amount of information being transferred between computers with the advent of Internet. Internet has now become the major medium of communication for people all over the world. Unfortunately, criminals are just as quick to exploit new technologies as any other section of the people. They are increasingly relying on the Internet for communication and exchange of information pertaining to unlawful activity. Consequently law enforcement agencies need to monitor the data flowing across the net to detect and prevent such activities. Companies that want to safeguard their recent developments and research from falling into the hands of their competitors also resort to intelligence gathering. Monitoring tools are also useful in evaluating and diagnosing performance problems of servers and network components. Monitoring tools should however, not compromise the privacy of individuals whose network communications are being monitored.

With the increase in use of computers and networks, the bandwidth at network backbones and Internet Service Providers is also increasing rapidly. For monitoring traffic at such busy segments of the network, there is a need for monitoring tools that can work at gigabit speeds without losing any relevant information. Such tools are also useful for monitoring Gigabit Ethernet LANs.

1.1 Sniffers

Network sniffers are software applications that are often bundled with hardware devices and are used for eavesdropping on network traffic. Sniffers usually provide some form of protocol-level analysis that allows them to decode the data flowing across the network according to the needs of the user. A sniffer may be used to understand and fix problems in network traffic or to detect abnormal activities, and unfortunately one may also be used by an attacker to steal critical information.

Sniffers on a LAN often means monitoring the traffic on the Ethernet. Ethernet was built around a shared principle: all machines on a local network share the same wire. Ethernet card (the standard network adapter) is hard-wired with a particular MAC address and ignores all traffic not intended for that address. The primary mechanism of sniffing in Ethernet is by putting the Ethernet hardware into "promiscuous mode" that turns off the filtering mechanism of the hardware chip on the network adapter and causes it to collect all frames irrespective of the destination MAC address. In a switched network, all machines do not receive all the packets as the switch sends a packet on only one outgoing port depending on the destination MAC address of the packet . Most switches allow "port mirroring" where a port can be configured as a "monitor" or "span" port that will get a copy of some or all of the traffic going across the switch. These ports can be used by sniffers. Alternatively, Ethernet taps can be used that allow us to examine network traffic without causing any data stream interference.

The amount of information that flows across the network is very high. A simple sniffer that just captures all the data flowing across the network and dumps it to the disk soon fills up the entire disk if placed on a busy segment of the network. Analysis of this data for different protocols and connections also takes considerable time and resources. Moreover, it would be desirable to gather data so that the privacy of individuals who are accessing and dispensing data through the network is not compromised. It is therefore necessary to filter, on-line, the data gathered by the sniffer.

Current day sniffers often come coupled with a filter that can filter packets based on various criteria. Three levels of filtering can be applied on these packets. The

first level of filtering is based upon network parameters like IP addresses, protocols and port numbers. This level of filtering is generally supported at the kernel level also. The second level of filtering is based on application specific criteria like email-id for SMTP, hostname for HTTP etc. The third level of filtering is based on the content present in the application pay load. Sniffers also come bundled with their own post-capture analysis and processing tools which extract information from the dump and present it in a human-readable form.

Several commercially and freely available sniffers exist currently. Sniffers come in different flavors and capabilities for different Operating Systems. Ethereal [4] and WinDump [2] are two such popular tools for Windows. On UNIX sniffers are generally based upon libpcap and/or BPF [10] (Berkeley Packet Filter). Libpcap is a standard packet capture library used to store packets on the disk. Many commercial and free post-processing and rendering tools are available that can analyze the packets stored by sniffers in the pcap format. BPF is an in-kernel packet filter that filters packets based on a directed acyclic Control Flow Graph method. BPF uses an interpreter for executing the filter code that assumes a pseudo machine with simple functionality akin to assembly language. Two popular sniffer tools on Unix are tcpdump [7] and Ethereal [4]. Tcpdump is based on libpcap and BPF filters. WinDump is a version of tcpdump for Windows that uses a libpcap-compatible library called WinCap.

Carnivore [5, 6, 14] is a network monitoring tool developed by FBI. It can be thought of as a tool with the sole purpose of directed surveillance. This tool can capture packets based on a wide range of application-layer based criteria. It functions through wire-taps across gateways and ISPs. Carnivore is also capable of monitoring dynamic IP address based networks. The capabilities of string searches in application-level content seems limited in this package. It can only capture email messages to and from a specific user's account and all network traffic to and from a specific user or IP address. It can also capture headers for various protocols.

PickPacket is a network monitoring tool that can address the conflicting issues of network monitoring and privacy through its judicial use. This tool has been developed as a part of the research project sponsored by the Department of Information

Technology, MCIT, New Delhi. The basic frame work for this tool and design and implementation of application layer filter for Simple Mail Transfer Protocol (SMTP) and Telnet has been discussed in Reference [9]. The design and implementation of application layer filter for Hyper Text Transfer Protocol (HTTP) and File Transfer Protocol (FTP) has been discussed in Reference [12]. The design and implementation of text string search in MIME-Encoded data has been discussed in Reference [1]. The design and implementation of application layer filter for the Remote Authentication Dial In User Service (RADIUS) Protocol has been discussed in Reference [8].

1.2 Need for Gigabit Sniffers

In the past few decades, use of computer networks for information exchange has increased rapidly. Also the number of users and the amount of information being transferred across the network have increased proportionately. With this surging demand for data, the bandwidth at the network backbones and Internet Service Providers has also increased. Bandwidth growth has been explosive in the Local area networks also, propelled by the availability and deployment of Gigabit Ethernet.

With this increase in bandwidth, a need for sniffers, that can monitor traffic at such high speeds, arises. A simple sniffer that captures all the data flowing across the network and dumps it to the disk soon fills up the entire disk especially if placed on a busy segment of the network. Moreover, it would be desirable to gather data flowing across the network so that the privacy of individuals who are accessing data through the network is not compromised. Thus it is necessary to filter on-line the data using various criteria. Filtering packets using complex criteria at very high speeds results in packet drops as packets arrive much faster at the interface card than they are handled by the sniffer. Once the buffers get filled, packets will be dropped at various levels starting from application to interface card. Thus there is a need for fast sniffers that can monitor traffic at high speeds based on complex set of criteria without dropping any packets.

Several commercial sniffers exist that claim to handle gigabit traffic. Sniffer Portable [15] by Network Associates and Unispeed Netlogger [17] are two such tools

developed for Windows. Reference [3] describes *ring sockets*, that can be used to improve the passive packet capture performance in Linux. nProbe/nFlow [11] is recently released for Linux that uses the technology described in Reference [3] to handle near gigabit sniffing. It provides accounting and performance information of a network by storing samples of traffic information in a standard flow format. Sniffing at gigabit speed on Linux is still not a matured technology.

In this work we have developed Gigabit PickPacket, a new version of PickPacket that can effectively use multiprocessor machines, cluster of machines and their combination to monitor gigabit traffic. Instead of just providing performance and accounting information of a network, Gigabit PickPacket can reconstruct the whole connection of interest without sacrificing the features provided in the original PickPacket.

1.3 Organization of the Report

This thesis focuses in detail on Gigabit PickPacket, a Network Monitoring Tool that can handle one gigabit per second traffic. Chapter 2 describes the high level architecture and design of the original PickPacket. Chapter 3 describes the design aspects of Gigabit PickPacket. Chapter 4 describes in detail various implementation issues and optimizations used in Gigabit PickPacket. Chapter 5 describes testing setup and performance results. The final chapter concludes the thesis with suggestions for future work.

Chapter 2

PickPacket: Architecture and Design

This chapter discusses the architecture and design of PickPacket. First, the architecture of PickPacket and its various components are described. Then design of each component is described briefly. Detailed design and implementation details are discussed in Reference [9].

2.1 Architecture

PickPacket can be viewed as an aggregate of four components ideally deployed on four different machines. These components are

1. *PickPacket Configuration File Generator* is a JAVA GUI deployed on a Windows/Linux machine. It is used to specify the criteria for capturing the packets. The criteria specified by the user are stored in a configuration file.

2. *PickPacket Filter*, deployed on a Linux machine, uses the configuration file as input, filters and stores the packets that match the specified criteria. Filtering is done based on criteria corresponding to IP addresses, port numbers, application layer protocol parameters and content present in the application payload.

3. *PickPacket PostProcessor*, deployed on a Linux machine, processes the packets stored offline and retrieves the meta information from them.

4. *PickPacket Data Viewer* is a GUI deployed on a Windows machine. It reads the meta information generated by the PostProcessor and displays it to the user.

An architectural view of PickPacket is shown in Figure 2.1 where these components are shown in rectangles.

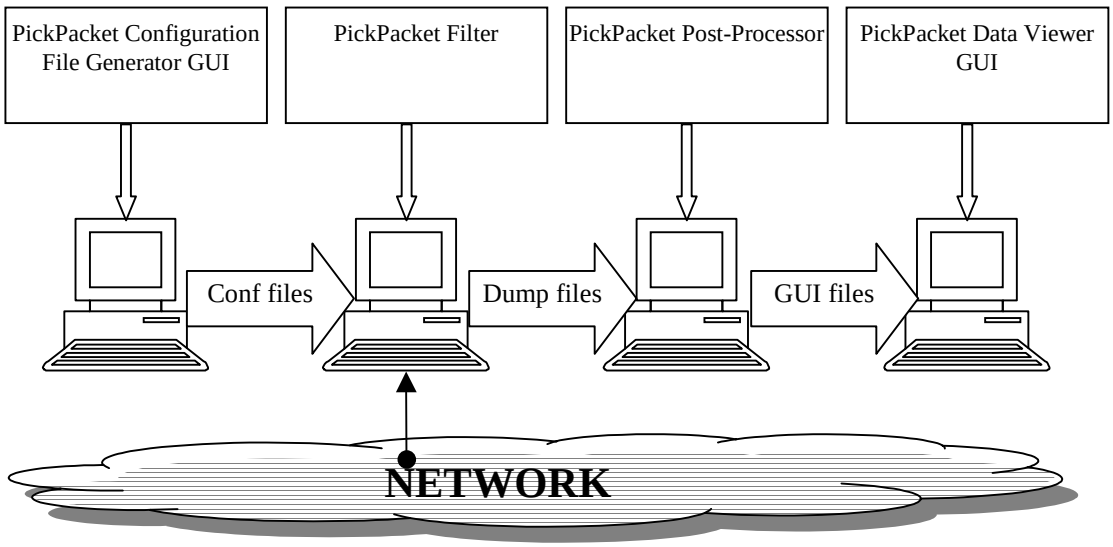


Figure 2.1: Architecture of PickPacket

2.2 Design

This section briefly describes the design of each component of PickPacket.

2.2.1 The PickPacket Configuration File Generator

The PickPacket Configuration File Generator is a Java based graphical user interface (GUI) that is used for specifying the rules for capturing the packets. These rules are saved in configuration file that is used as input for PickPacket filter. This file is a text file with HTML like tags. A sample configuration file is given in *Appendix A*. This file has four sections:

1. The first section contains specifications of the output files that are created by the PickPacket Filter for storing captured packets. There is no restriction on the number of output files. The last file can have a size of "0" meaning potentially infinite size. A feature in the configuration file is the support for different output file managers. This feature would be useful if captured packets have to be stored in formats other than the default pcap [18] style format.

2. The second section contains criteria for filtering packets based on source and destination IP addresses, transport layer protocol, and source and destination port numbers. The application layer protocol that handles packets that match the specified criteria is also indicated. This information is required for demultiplexing packets to the correct application layer protocol filter.

3. The third section specifies the maximum number of simultaneous connections that can be monitored for any application. This is used for memory allocation. The default value set by the configuration file generator is 500 for each application protocol. A very large value may cause the system to run out of memory, and thus behave unpredictably. A small value may cause some connections to be missed.

4. The fourth section comprises of multiple subsections, each of which contains criteria corresponding to an application layer protocol. Based on these criteria the application layer data content of the packets are analyzed. Filtering criteria for SMTP, HTTP, FTP and Telnet protocols can be specified in these subsections. An application layer protocol subsection also specifies the mode of operation of the filter ("PEN" or "FULL") for the protocol. In PEN mode, packet data till and including the transport layer protocol header is saved whereas in FULL mode, the entire packet is saved.

2.2.2 PickPacket Filter

PickPacket Filter reads packets from the network and processes them to find if they match any of the criteria specified by the user. If a match occurs, the packet is

saved onto the disk for further analysis. This section briefly describes the design of the PickPacket Filter.

The PickPacket filter can filter packets at three levels.

1. Filtering based on network parameters (IP addresses, port numbers, etc).
2. Filtering based on application layer protocol specific criteria (hostnames, email-ids, etc).
3. Filtering based on content present in the application payload.

The first level of filtering has been made very efficient through the use of in-kernel filters [10], as only packet which matches the network level criteria are copied from kernel space to user space. Since the content of application can be best deciphered by the application itself, the second and third levels of filtering are combined.

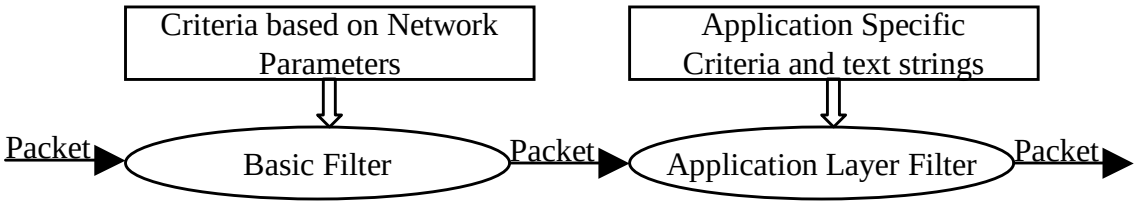


Figure 2.2: Filtering Levels

Figure 2.2 illustrates the various levels of filtering. Basic filter reads packets from the network and filters them based on the network parameters specified in the configuration file. It passes only those packets that satisfy the criteria to the next level. Application level filter further filters the packets received from basic filter based on application specific criteria.

Since it would be convenient to have a separate filter for each application layer protocol, application level filtering is split into multiple filters – one for each protocol. This design has the advantage that it is easy to enhance the filter by adding new application layer filters. A demultiplexer is provided between basic filter and

application level filters. It decides which application filter should receive the packet for further processing based on its own set of criteria.

Application specific filtering reduces to text search in the application layer data content of the packets. In case of communication over connection oriented protocol, this text search handles situations where the desired text is split across two or more packets before being transmitted on the network. It also handles the case where packets are received out of sequence. *TCP Connection Manager* is present between demultiplexer and application layer filters to find whether a packet is out of order. It is designed in such a way that it will handle only those connections that are of interest to the application layer filter. Application layer filter can alert it so as to maintain the sequence information for a connection.

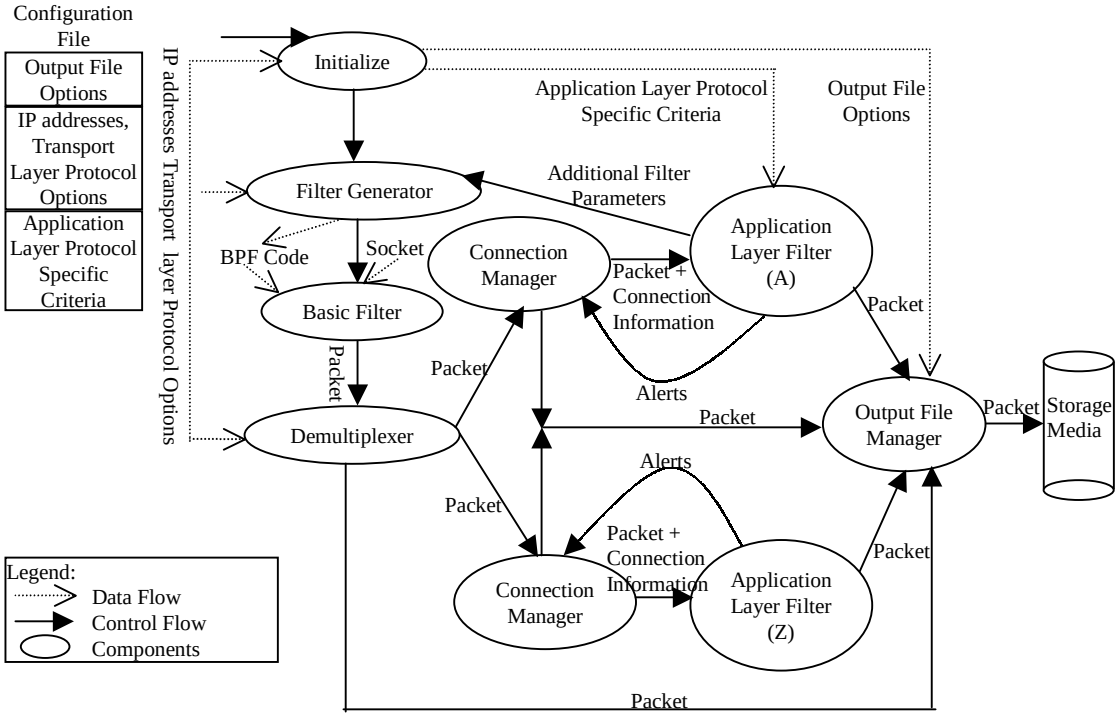


Figure 2.3: The Basic Design of the PickPacket Filter

Figure 2.3 shows the major modules in the PickPacket Filter. The module *Initialize* is used for initializations dependent on the configuration file. Another module,

the *Output File Manager*, is responsible for dumping filtered packets to the disk. The *Filter Generator* module is used for generating the in-kernel BPF code. Hooks are provided for changing the BPF code on-the-fly. Functions that can generate the filter code based on changed parameters can be called by application level filters such as FTP during "PASSIVE" mode of file transfers. The *Demultiplexer* can also call the *Output File Manager* directly so that the filter can directly dump packets without resorting to application layer protocol based filtering, if necessary. The *Connection Manager* can also directly dump packets to the disk. This is required when all criteria have matched for a specific connection and the connection is still open. More details of these components can be found in Reference [9].

The *output file manager* stores output files in the *pcap* [18] file format. This file starts with a 24 byte pcap file header that contains information related to version of pcap and the network from which the file was captured. This is followed by zero or more chunks of data. Every chunk has a packet header followed by the packet data. The packet header has three fields – the length of the packet when it was read from the network, the length of the packet when it was saved and the time at which the packet was read from the network. The data stored in pcap file format can also be viewed using utilities like tcpdump. This standard format also allow us to use other tools for analysis of captured data.

The PickPacket Filter contains a text string search library. This library is extensively used by application layer filters in PickPacket. This library uses the *Boyer-Moore* [13] string-matching algorithm for searching text strings. This algorithm can be used for both case sensitive and case insensitive search for text strings in packet data.

2.2.3 The PickPacket Post-Processor

The PickPacket PostProcessor processes the packets stored by the filter in a dumpfile offline and separates the packets based on transport layer and application layer information. The detailed description of Post Processor is given in Reference [9].

The Post Processor has three components – the *Sorter*, the *Connection Breaker*, and the *Meta Information Gatherer*. These are shown in Figure 2.4.

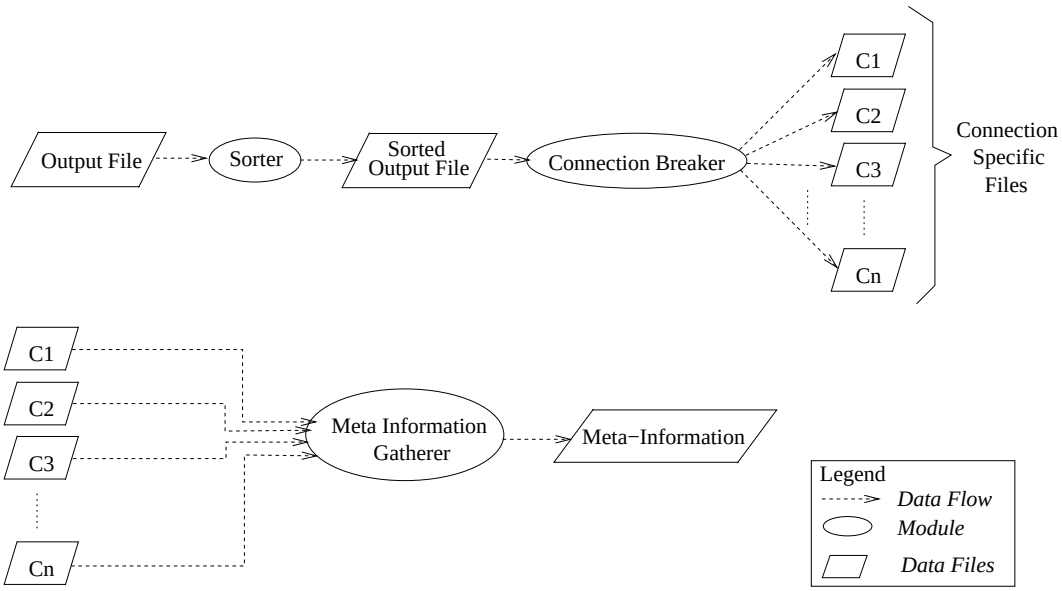


Figure 2.4: Post-Processing Design

The packets present in the output file may not be in the order they were transmitted on the network. Therefore the *Sorter* module is used to sort the packets present in the output file generated by the packet filter based on the time stamp value corresponding to the time the packets were read off the network. The *Connection Breaker* module reads the sorted output file and retrieves the connection information from the packets belonging to a connection oriented protocol and separates them into different files. Internally connection breaking is accomplished by a TCP state machine based process. Packets belonging to a connectionless protocol like UDP are separated based on the communication tuple. The *Meta Information Gathering Module* reads these connection specific files and retrieves the meta-information of every connection. Each application requires different meta-information and packets belonging to a particular application are processed by meta-information gathering modules for that application. The meta-information of application layer protocols includes important fields present in the data content such as e-mail addresses for SMTP connections, usernames for FTP connections, URLs for HTTP, etc. The meta-information for different application layer protocols is stored separately.

2.2.4 The PickPacket Data Viewer

The PickPacket Data Viewer is used for rendering the post-processed information. This is a Visual Basic based GUI and runs on Windows. The choice of this platform was made for rapid prototyping and the rich API (Application Program Interface) library that is provided in Windows for rendering content belonging to an application. The Data Viewer reads the meta-information files and lists all connections by application type, source and destination IP addresses, and other such fields based on the meta-information that has been provided by the Post-Processor. These connections can be sorted and searched based on these fields. The Data Viewer also allows examining the details of a connection and can show the data for that connection through appropriate user agents commonly found in the Windows environment such as Outlook Express, Internet Explorer, etc. The dialogue between communicating hosts can also be seen in a dialogue box. User can also view the configuration file used by the packet filter.

Chapter 3

Design of Gigabit PickPacket

PickPacket can filter packets based on network and TCP/UDP level criteria as well as application level criteria for SMTP, FTP, HTTP and Telnet protocols. PickPacket also supports monitoring dialup users who are allocated dynamic IP addresses by the Internet Service Provider with the RADIUS support included in it [8]. The major earlier version of PickPacket was designed for operation at 100 Mbps. It can not handle packets when they are received at a very high speed. When the packets are arriving at a very high speed, PickPacket filter that runs on-line should process the packets fast enough in order to avoid dropping any packet. Analyzing the packet for specified criteria and storing the packet to a file takes more time than the rate at which packets arrive in gigabit networks. So, the PickPacket Filter starts dropping the packets. This problem can be tackled by using a parallel architecture, where filtering can be parallelized. But the current design does not support this feature and thus it does not scale to gigabit networks.

Gigabit PickPacket is an enhanced version of PickPacket that induces parallelism into the filtering component of the PickPacket thus enabling it to monitor gigabit traffic. This chapter discusses the design of this tool. Two mechanisms for achieving parallelism are described in the rest of this chapter.

3.1 Multi-threaded Design

Multi-threaded and distributed computing are gaining a wide popularity in the area of high performance computing. Availability of high speed computer networks and sophisticated software environments are allow performing parallel/concurrent computing on commodity hardware. Recently, threads have become powerful entities to express parallelism on these shared memory multiprocessors (SMP) systems. On a multiprocessor machine, multiple threads may be distributed across multiple processors, which can dramatically improve throughput. This is often the case with powerful multiprocessor web servers, which can distribute large numbers of user requests across CPUs in a program that allocates one thread per request. These factors have given an impetus for further popularity of multi-threading.

In the past, high-performance multi-threading has been used only in supercomputing, real-time control and multi-user server applications for achieving high throughput. The idea of dividing a computationally intensive program into multiple concurrent threads to speed up execution on multiprocessor computers is well established. However, this kind of high-performance multi-threading has made very little impact in mainstream business and personal computing, or even in most areas of science and engineering. The reason has been the rarity and high cost of multiprocessor computer systems. With the advent of inexpensive multiprocessor PCs, multi-threading is poised to play an important role in all areas of computing.

PickPacket filter is the most crucial component of the PickPacket architecture that monitors traffic on-line at a very high speed. To handle such traffic, the filter should take decision about a packet as soon as possible. Thus, multi-threading is introduced in the filter component to enable it to use multiprocessor support. Even though it is clear that multi-threading will improve the performance, it is not a trivial exercise to convert PickPacket into a multi-threaded application.

The PickPacket Filter can be parallelized in different ways:

1. We can have one thread for each application layer protocol and all the packets corresponding to this application protocol can be handled by this thread. This method is easy to implement but is not efficient when packets belonging to one application protocol dominates the other application protocols, which is quite

common in practice. In that case, work load will not be uniformly distributed across all the threads. In real life, HTTP traffic is more than the combined traffic of all the other application protocols.

2. We can create fixed number of threads and allocate packets to threads in round-robin manner or using any other load balancing algorithm. In this case, packets belonging to the same connection may be handled by different threads. In that case, we need to protect connection specific data structures and application level information stored for a connection from race condition. We can use locks to achieve this, but this is not an efficient design as these data structures are used so often in filter that efficiency achieved by using multiple threads will be compromised by the contention for locks.

3. We can use another method where one thread is created for each new connection that handles all the packets belonging to this connection and destroyed when the connection is closed. In this method thread management overhead and context switches will be significant and nullify the advantage of using multi-threading.

As all these methods have some bottlenecks, we use a different approach where a fixed number of threads are created and instead of distributing individual packets among these threads, different connections are distributed among them in the desired ratio. All packets belonging to a particular connection will always be handled by the same thread. This design achieves load balancing at a much finer level than approach 1 above, as packets are distributed based on connections rather than on application layer protocol. Also it solves the contention problem of approach 2 as we need not protect all the connection specific data structures from race condition, because each connection is always handled by the same thread. Problems in approach 3 will not be present here as we use a fixed and small number of threads. Ideally this number should be equal to the number of processors on this machine for achieving the best performance.

We use a hash function on four tuple (Source IP Address, Destination IP Address, Source Port and Destination Port) to distribute packets among multiple threads in

the desired manner. Four tuple is chosen for calculating hash function because all packets belonging to a connection have the same four tuple. We categorize threads in our model into two different types based on the task that they perform. They are *reading threads* and *processing threads*. A reading thread reads packets from the network and handles it or enqueues it for other thread to handle based on the hash function. A processing thread on the other hand does not read any packet but processes packets that are read by other threads. Each processing thread maintains a buffer called *pending queue* in which packets are inserted by the reading threads on hash index match. As packet reading time is lesser than packet handling time, generally we require fewer reading threads compared to processing threads. We can use all reading threads and no processing threads if the load is to be shared equally among all the threads.

The Psuedocode for reading thread follows.

```
reading_thread
{
        while(1)
        {
                read a packet from the network;
                search dynamic demultiplexer table for entry;
                if(entry found)
                {
                        thread_index = thread index present in the entry;
                        insert into pending queue of target thread whose
                          index range matches thread_index;

                        if target thread is a processing thread and its
                          pending queue was empty before insertion then
                          send signal to that thread;

                        continue;
                }
```

```
                search static demultiplexer table for entry;
                if(entry found)
                {
                        thread_index = hash(four tuple);
                        insert into pending queue of target thread whose
                          index range matches thread_index;

                        if target thread is a processing thread and its
                          pending queue was empty before insertion then
                          send signal to that thread;

                        continue;
                }
                else
                        discard packet;

                while(pending queue is non-empty)
                {
                        remove packet from pending queue;
                        process_packet();
                }
        }
}
```

Packets after being copied from the network are checked against the criteria based on the application layer data content present in them. For this the packet filter determines to which application layer protocol the packet belongs and passes it to the respective filtering module. In other words packets are demultiplexed on the basis of the application layer protocol they belong to. We use demultiplexer tables for maintaining this information. Each table contains tuples representing the basic criteria specified in the configuration file. The packet is sent to the appropriate connection manager and application filter based on the information in these tuples.

There can be a situation where an application might require addition of new tuples in the demultiplexer table apart from the tuples corresponding to the basic criteria specified in the configuration file. An example of such a situation is *passive FTP*. In a passive file transfer, the FTP client contacts the server on the standard FTP command port and issues a PASV command. The FTP server replies with its own IP address and a port to which the client is supposed to connect for data connection. This port when sent from the server is a non-standard port and hence cannot be determined beforehand. For monitoring such connections, the in-kernel BPF code i.e., the BPF code, needs to be modified.

For this purpose the demultiplexer maintains its tables in two separate parts, a *static table* and a *dynamic table*. The static table contains information about the basic criteria specified in the configuration file. Whenever an application layer protocol filter desires a modification in the BPF code, it adds a new entry into the dynamic demultiplexer table and removes the existing BPF code from the kernel.

On reading a packet from the network, a reading thread first searches in the dynamic demultiplexer table for an entry corresponding to this packet. If an entry is found, packet is inserted in the pending queue of the appropriate thread. If an entry for this packet is not found in the dynamic demultiplexer table, the static demultiplexer table is searched. If an entry is found in this table, packet is inserted in the pending queue of the appropriate thread whose index is calculated by applying hash function on the four tuple of this packet. After inserting packet in pending queue of the other thread, the target thread is signaled if it is a processing thread and its pending queue was empty before this insertion. Signal is not sent for other reading threads and processing threads with non empty pending queues as they will eventually check their pending queues and process this packet. If the entry is not found even in the static table then this packet is discarded. Now if pending queue of current thread is non empty, packets are dequeued and processed until it becomes empty.

Dynamic demultiplexer entries use different thread index for packet handling than the one calculated by applying hash function on four tuple of the packet. In FTP, control and data connections share the same data structures throughout the

application filter processing. To avoid contention here, FTP control and data connections are handled by the same thread. When an FTP control connection is being handled, FTP filter adds an entry in the dynamic demultiplexer table corresponding to the FTP data connection. Index of the thread handling control connection is added in the dynamic demultiplexer table along with the basic criteria information. This index is later used to deliver the data connection to the same thread that handled the control connection.

The Psuedocode for processing thread follows.

```
processing_thread()
{
        while(1)
        {
                if(pending queue is empty)
                {
                        wait for signal;
                        On receiving a signal continue;
                }
                else
                {
                        remove packet from pending queue;
                        process_packet();
                }
        }
}
```

A processing thread waits for signal on finding that its pending queue is empty. If the queue is not empty, it removes a packet from the queue and processes it. After processing, it will again check for any packets in the pending queue. On receiving a signal in the waiting state, it checks for packets in pending queue.

This model gives fine control over dividing the load among different threads in the desired ratio. Also, it is scalable for any number and type of processors. Fine tuning

of ratios, number and type of threads is required to achieve maximum performance for given hardware.

3.2 Distributed Design

Distributed computing solves a large problem by dividing it into small problems, solving them at many computers and finally combining the partial solutions into a solution for the original problem. Recent distributed computing projects have been designed to use the computers of hundreds of thousands of volunteers all over the world connected through Internet for solving many computationally intensive problems. Distributed computing can be effectively used to get the most out of multicomputer systems in solving computationally intensive problems. MPI/PVM is used for message passing between different computers.

This technique is used in Gigabit PickPacket to effectively use the power of multi-computer systems in monitoring gigabit traffic. This section describes two variations of our design in using distributed computing.

3.2.1 Hub based approach

In this approach, multiple machines run the monitoring tool in parallel and computation overhead is distributed among these machines in a desired ratio. We do not need any message passing between these machines. We can either use single threaded or multi-threaded PickPacket at each machine. At each machine, only a subset of the incoming packets are handled and the remaining packets are discarded. A hash function calculated on the four tuple (Source IP Address, Destination IP Address, Source Port, Destination Port) of the packet is used to find this subset. Hash function is chosen in such a way that all packets belonging to a connection are handled by the same machine and a packet is not discarded by all the machines. Hash index generated by the hash function lies in the index range of one and only one machine. Figure 3.1 shows the architecture of hub based approach.

For handling FTP and RADIUS protocols, some deviations from the original design are required. In Passive FTP, four tuple of the data connection is known

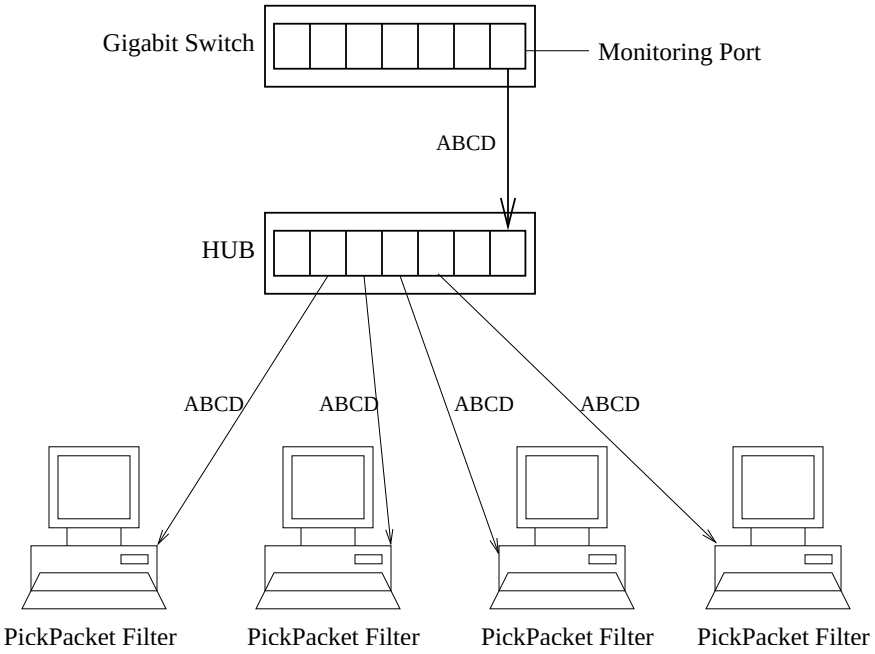


Figure 3.1: The Architecture of Hub based approach

only at the time of handling control connection. FTP filter dynamically changes the BPF code and recompiles it to prevent in kernel filtering of data connection packets by Linux Socket Filter. It also adds an entry in the dynamic demultiplexer table for the new data connection. The BPF code and dynamic demultiplexer entries are changed only on the machine where corresponding FTP control connection is being handled. Thus FTP data packets reach the application layer, without being filtered out by the BPF filter, only on this machine and they should be handled by this machine without filtering based on the hash function. Because of these reasons, FTP should be handled with special care to prevent hash function from discarding data connection packets on the machine where control connection is being monitored.

In RADIUS protocol, authentication and accounting packets are the control packets that will instantiate new connections. All the packets corresponding to a RADIUS control connection should be handled by the same machine as we maintain state information corresponding to the control packets received. But these packets may be having different four tuples as authentication and accounting servers run on

different ports and possibly on different machines. Thus RADIUS control packets should be exempted from hash function and special care should be taken while handling them. Similar to FTP, RADIUS control packets add entries in demultiplexer table and recompile BPF code to enable monitoring of RADIUS instantiated connections. So, packets belonging to RADIUS instantiated connections will reach the application layer only on the machine where RADIUS control connection is handled. Thus, RADIUS data packets should not be discarded on this machine based on hash function and they should be handled similar to FTP data packets.

To solve the above problems, FTP data packets and RADIUS packets should be handled before discarding them based on hash function. Figure 3.2 shows the data flow diagram of Gigabit PickPacket with special care for FTP and RADIUS.

Every UDP packet is first checked for whether it is a RADIUS packet. If it is a RADIUS packet, it is handled without calculating hash function. Thus, RADIUS control packets are handled by all the machines and BPF Filter on every machine is changed to accept packets belonging to RADIUS instantiated connections. Dynamic demultiplexer table is divided into FTP and RADIUS demultiplexer tables to avoid searching RADIUS entries of demultiplexer on every machine for all the received packets. On receiving a TCP packet, FTP demultiplexer table is searched for an entry corresponding to this packet. If an entry is found, this packet is handled without discarding based on hash function. Otherwise RADIUS demultiplexer table and static demultiplexer table are searched in order for an entry corresponding to this packet. If it is found in any of these tables, it is handled normally and discarded otherwise. With this approach, RADIUS instantiated connections are distributed among all the machines based on the same hash function.

3.2.2 Splitter based approach

Even though hub based approach improves the performance of PickPacket to some extent, it is still limited by the kernel level overhead as all packets are handled by the kernel on every machine. The Hub based approach distributes the application level overhead but not the kernel level overhead. In this section, we discuss another approach where kernel level packet overhead can also be distributed.

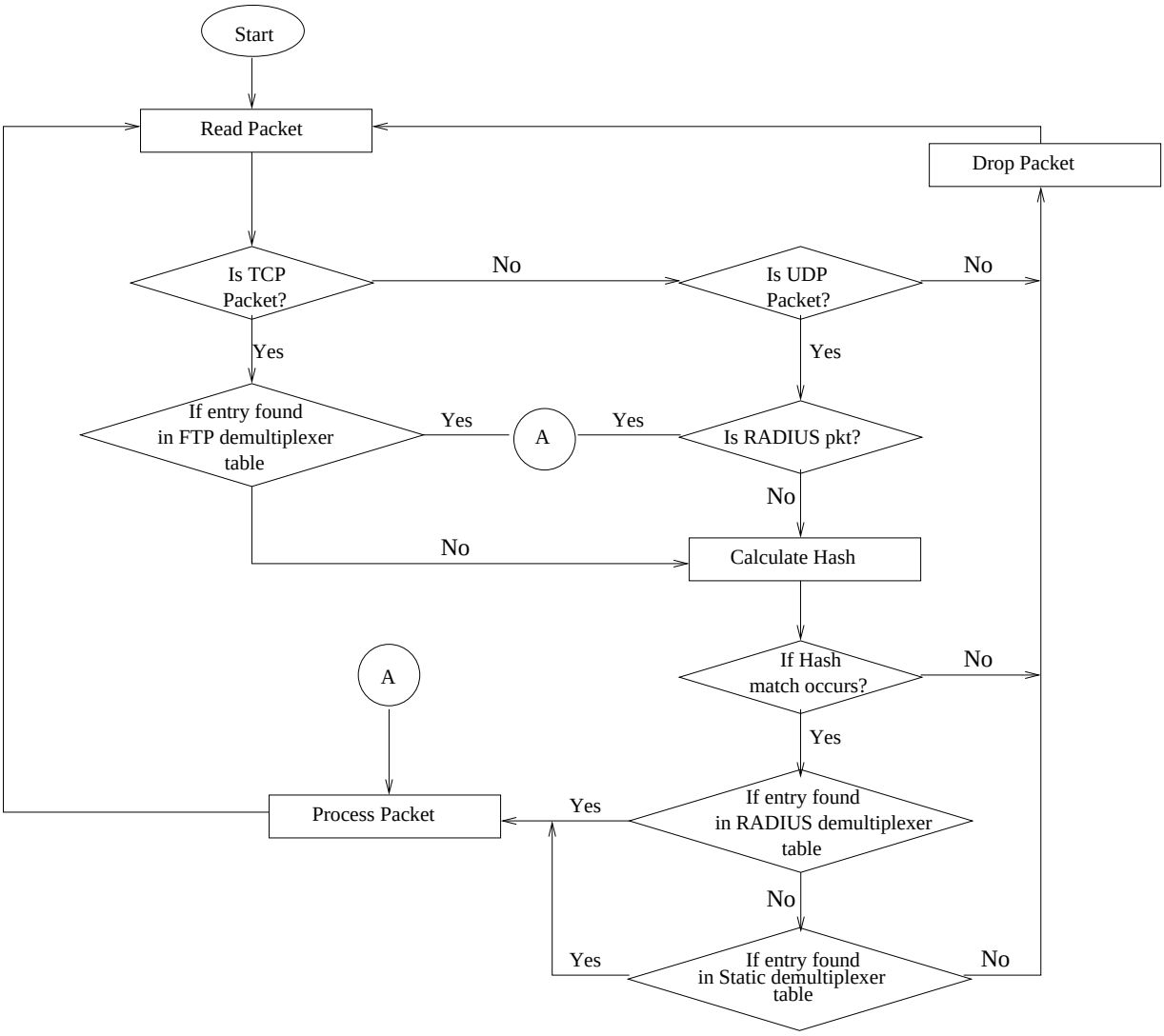


Figure 3.2: Packet handling in Hub based approach

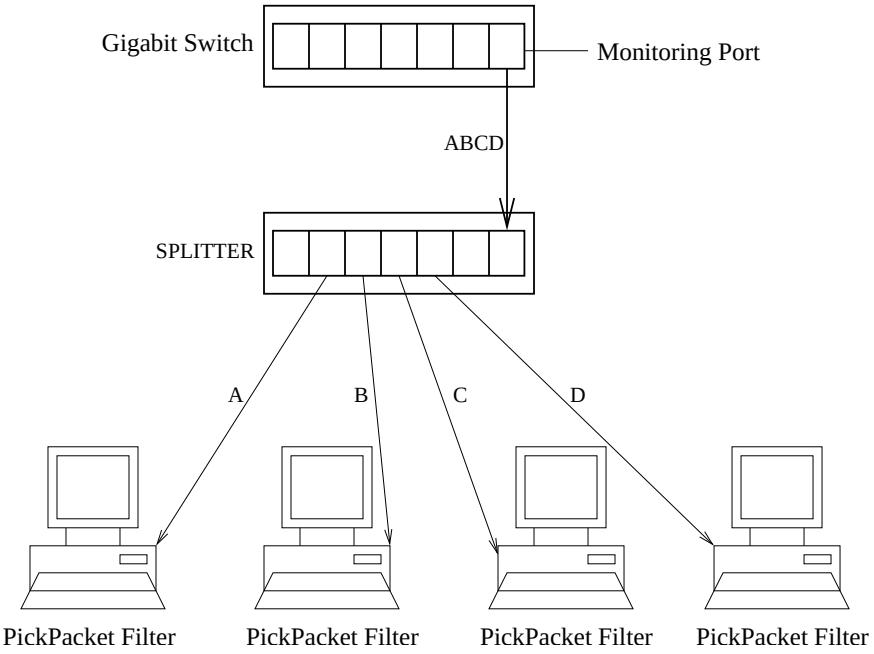


Figure 3.3: The Architecture of Splitter based approach

Load balancing switches can be used to split the traffic to be monitored among multiple machines in such a way that all packets belonging to a connection are sent to the same machine. Figure 3.3 shows the architecture of splitter based approach. IDS load balancers are available [16] that can split traffic based on round robin and weighted least connections algorithms. We can use the architecture shown in the figure, where a load balancing switch will divide the traffic to be monitored among multiple machines running PickPacket. By this approach kernel level overhead is also distributed. This approach is scalable to very high speed networks.

This approach has the disadvantage that FTP and RADIUS can not be handled. As we have already seen, a simple hash function that discards all the packets that do not match the hash index is not enough for handling FTP and RADIUS protocols. Special care need to be taken to handle FTP control and data connections at the same machine. Similarly RADIUS control and data packets should be handled by the same thread. A general load balancing switch may not handle these variations. A customized splitter can be built with special care to handle these protocols.

Various approaches for parallelizing the filter component of Gigabit PickPacket have been discussed in this chapter. Multi-threaded approach effectively uses the power of multiprocessor machines to distribute the application load among multiple processors. It allows binding a thread to a particular processor and controlling the load on various threads for achieving the best performance. The Hub-based approach distributes the application load among multiple machines in a desired ratio and thus improves the performance of filter. It is used when multiple machines are available for monitoring the traffic. Multi-threaded and hub-based approaches can be used together when multiple multiprocessor machines are available. Both these approaches distribute the application load but not the kernel load among multiple processing units. So, they are useful when complex criteria are used for filtering, where application overhead is more than kernel overhead. In both these approaches, as all packets are handled at the kernel level, kernel limitation in handling maximum traffic speed limits the maximum speed that can be monitored by Gigabit PickPacket.

In the splitter based approach, kernel level packet handling is also distributed among multiple machines. So, Gigabit PickPacket with this approach is not limited by the kernel limitation in handling high speed traffic. The only limitation of this approach is that hardware splitter needs to be customized to support FTP and RADIUS protocols.

Current implementation of Gigabit PickPacket includes support for all three approaches. By changing the configuration specification, any combination of these approaches can be obtained. Uniprocessor version can be obtained by setting number of threads and hash index to one.

Chapter 4

Implementation of Gigabit PickPacket

This chapter discusses the implementation details of Gigabit PickPacket. First the enhancements made to configuration file generator are described. Then the implementation of filter and changes for postprocessing are described briefly.

4.1 Configuration File Generator

The configuration file generator of PickPacket is enhanced so that users can also specify the criteria for multi-threading and hash function calculation. For multi-threaded approach, configuration file should contain thread-specific information such as load on each thread, type of thread, its processor binding and pending queue length. Load on each thread can be varied by user to reduce the load on a thread running on a processor that performs kernel level packet handling. Threads can be bound to processors to prevent unnecessary context switches that result in unpredictable overheads. The pending queue length of each thread needs to be controlled for best performance. A very high value will cause the system to run out of memory, whereas a very small value may cause dropping of packets at the pending queue.

For the hub-based approach, hash index values need to be specified in the configuration file. Users should be able to control the load on various machines and

threads in any ratio. We use the total hash index to represent the total load on the system and hash start and end values for each machine represent the hash index range of the current machine. If the hash value calculated by the hash function falls in the hash range of a machine, then it will handle the packet.

A new section is added in the configuration file before the application level criteria sections called thread_info. This section contains information about hash_index, num_threads, hash_start_value, hash_end_value and subsections containing information specific to each thread. Hash_index is the value of total load that is being shared by all threads on all the machines, num_threads is the number of threads that are going to be created on this machine, hash_start_value and hash_end_value specify the range of hash index for this machine. A subsection is created for each thread that contains information about type of thread, load on this thread, processor binding and length of the pending queue. Appendix A shows a sample configuration file for Gigabit PickPacket.

A panel called Thread Manager Panel is added in the Configuration File Generator GUI for specifying the new criteria. Figure 4.1 shows the new GUI screen for specifying the criteria added in Gigabit PickPacket.

According to the specifications in Figure 4.1, this is one machine in the cluster that takes half of the total load as its hash range is half of the total hash value. Two threads are created where one thread is a reading thread binded to processor 0 that reads all the packets but processes only 25% of them and the other thread is a processing thread binded to processor 1 that handles 75% of the packets read by reading thread with a pending queue length of 1000 packets.

4.2 Filter

This section discusses the implementation details of various design methodologies explored in the previous chapter. Some optimizations in the filter component and digital signature implementation for protecting dumpfile's integrity are also described briefly.

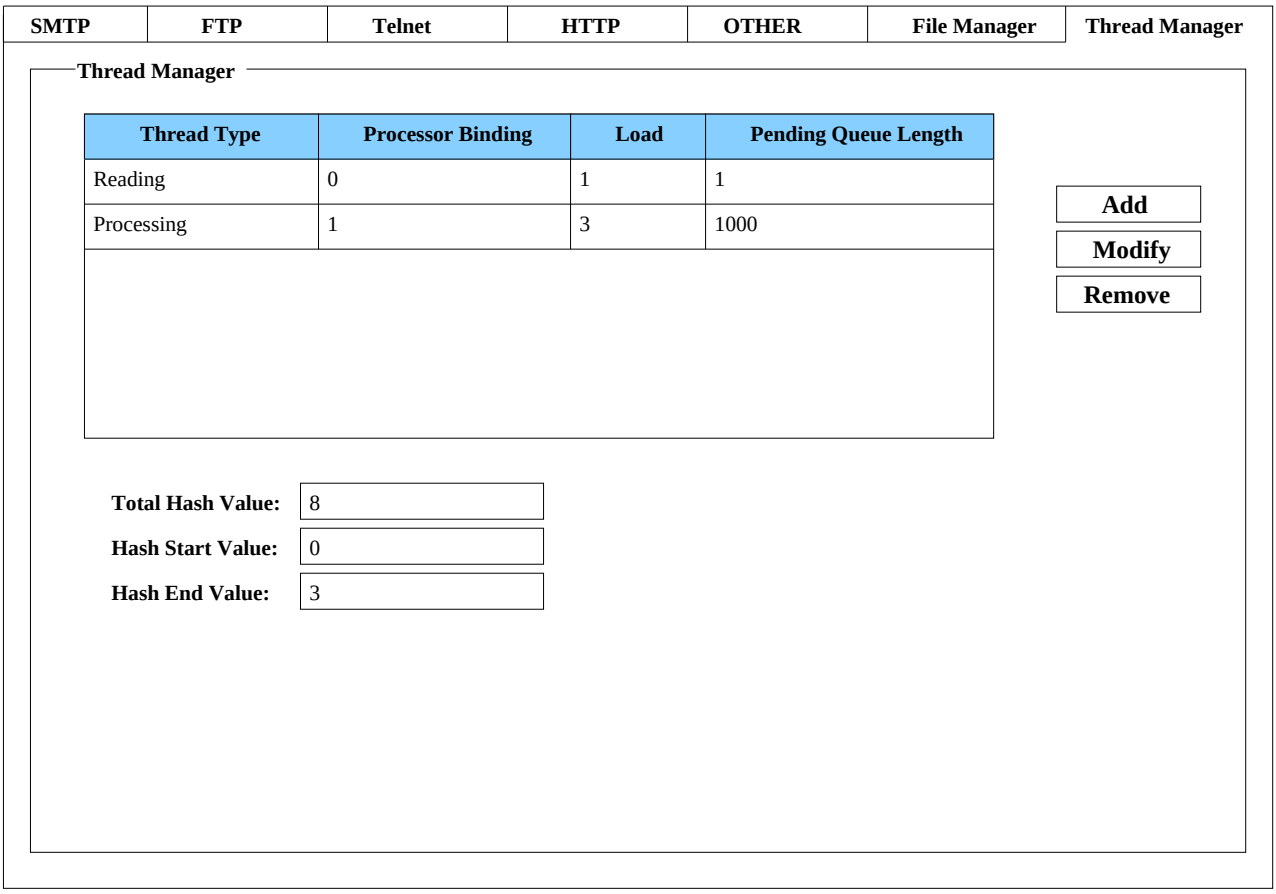


Figure 4.1: Configuration File Generator: Thread Manager Tab

4.2.1 Multi-threaded Implementation

We used standard POSIX thread library on Linux for creation and management of threads. The Filter reads thread-specific information from the configuration file and creates as many threads as specified in the configuration file. It starts reading packets from the network only after all the threads are properly initialized. Each thread uses a structure called *thread_specific_data* for maintaining thread specific information such as pending queue pointers and information about current packet being handled by this thread.

As we have multiple threads running simultaneously and sharing some global data structures, we need to protect these data structures from race conditions. Demultiplexer tables, tcp active and free lists, application level free lists are some global data structures shared by all threads that need to be protected from simultaneous access by multiple threads. Each thread spends very little time executing the code in critical sections involving these data structures as the critical sections are small and each thread is ideally scheduled on its own processor. So, we use spinlocks for protecting these critical sections rather than blocking locks. Non-blocking read/write locks are also implemented on top of basic spinlocks for protecting data structures like dynamic demultiplexer, where reads are frequent and writes are rare.

4.2.2 Clustered Implementation

In clustered implementation, traffic should be distributed among all the machines running PickPacket in the desired ratio and all packets belonging to the same connection should always be handled by the same machine. The hash function is calculated on the four tuple (Source IP Address, Destination IP Address, Source Port, Destination Port) of each packet as all packets belonging to a connection have the same four tuple. As it is possible to have different machines with different power, we need to divide the load between these machines any desired ratio. For providing this, we used hash index range for each machine and thread instead of single hash index value. Hash value is calculated as sum of all the items in four tuple modulo total hash index range of all the machines. Only that machine whose hash range includes this value accepts this packet, while all the other machines discard it except for some

variations in FTP and RADIUS. This method distributes all the connections in the desired ratio as the hash is calculated on connection information basis.

4.2.3 Signature for Dumpfiles

For providing authenticity and integrity to dumpfiles, a digital signature is generated for each dumpfile generated by the filter. A message digest is a special number that is effectively a hash code produced by a function that is very difficult to reverse. A digital signature is a message digest encrypted with someone's private key to certify the contents. This process of encryption is called signing. This digital signature can later be decrypted using a publicly known key to verify that it is signed with this private key.

The most common digital signature in use today is the combination of the MD5 message digest and the RSA encryption. We used this combination to generate digital signature for all the dumpfiles generated by filter. Incremental MD5 is used to generate message digest when each packet is stored to dumpfile and the message digest is finally encrypted while closing dumpfile. Private key is input to the filter process by a safe medium such as a removable disk.

The digital signatures thus generated for dumpfiles are sent to the postprocessor where verification of message digests is done. Digital signature is decrypted using a public key, that is known to the world, to generate the message digest. Message digest of the dumpfile is calculated now and it is compared with the decrypted message digest. If a mismatch occurs between them, it means that the dumpfile has been changed before postprocessing and thus its integrity is lost. This mechanism of digital signatures thus protects the integrity of dumpfiles.

4.2.4 Optimization of Filter

Filter can be optimized by using memory mapped I/O on sockets for reading packets. This will reduce the packet reading time by eliminating a memory copy from kernel space to user space. A ring buffer of memory is allocated and attached to the raw socket and the socket is configured so that kernel will use this ring buffer for

reading packets into memory for this socket. The same buffer is shared by kernel and application for processing the packet. After processing the packet, application sets a flag in this packet indicating that this memory space can be reused by kernel. The application should process the packets fast enough to prevent kernel from dropping packets due to lack of empty frames in the ring buffer. As mmap saves a memory copy for all the packets reaching the application layer, it reduces the overall kernel level packet handling overhead by a major factor.

Latest Ethernet drivers compiled with NAPI support are used in machines running PickPacket for better performance. NAPI is a device polling technology introduced from Linux 2.4.20 that controls the interrupt rate by polling the device for packets at regular intervals thus improving the performance of operating system in handling high traffic rate. NAPI reduces the kernel overhead in handling interrupts by polling for interrupts at regular intervals rather than device sending an interrupt after receiving packets. When large number of packets have been received, multiple packets can be handled in a single poll in an efficient manner. Kernel level packet handling overhead is reduced a lot by using this technique as the time spent in handling interrupts for each packet is not present here.

Another optimization is to prevent the sniffed packets from reaching the TCP/IP stack of the machine running Gigabit PickPacket as the sniffed packets are not destined for applications running on this machine. This further reduces the load on kernel level packet handling as expensive operations like IP checksum calculation and routing table lookup are eliminated by not allowing a packet from reaching TCP/IP stack on the machine. This can be accomplished using a kernel module that acts as an IP_PRE_ROUTING_HOOK in the Linux kernel.

4.3 PickPacket PostProcessor

PickPacket PostProcessor has been changed to postprocess multiple dumpfiles generated by various machines in Gigabit PickPacket. A new program is added to postprocessor before the sorter module that will verify the integrity of individual dumpfiles and concatenate them into a single dumpfile. This single dumpfile is

given to sorter program for further postprocessing. For verification of signature, we recalculate message digest for each dumpfile using MD5 and compare this message digest with the one obtained by decrypting the signature for this dumpfile using public key. If both the message digests do not match, then the post processing stops, giving an error message. Otherwise, it checks all the dumpfiles for integrity and finally concatenates all of them into a single dumpfile. While concatenating, 24 byte pcap header is removed from all the dumpfiles except the first one. If the pcap output files are generated using different versions of the pcap library, then an error message is generated and the dumpfiles should be postprocessed separately.

Chapter 5

Testing

In this chapter we describe the test setup used for testing Gigabit PickPacket. The essential idea of these experiments was to determine the peak bandwidth at which Gigabit PickPacket monitors the network without dropping any packet. Performance is evaluated by specifying complex set of application level criteria and various traffic patterns were monitored with these criteria.

5.1 Correctness Testing

Functional testing of Gigabit PickPacket was carried out by varying the number of reading threads, handling threads and by testing all the control paths of application level filters with various criteria. Effect of multiple threads and usage of locks were thoroughly tested for correctness. For testing Gigabit PickPacket at high speeds with live traffic, a gigabit hub was required to sniff the packets by putting the interface in promiscuous mode. Due to unavailability of a gigabit hub, we changed the Linux kernel to set the destination MAC address of every outgoing packet to the Ethernet broadcast address. At the receiver, we changed the kernel to receive these MAC broadcast packets and send upto application layer without dropping them. We used three machines with 2.4 GHz CPU, 256 MB RAM running the changed Linux kernel connected through a gigabit switch as both clients and servers for various application layer protocols. Gigabit PickPacket was tested on two dual

processor Xeon machines with 2.0 GHz CPU, 1.0 GB RAM and running the Linux kernel 2.4.20-8smp connected to the same switch as traffic generating machines. We used some scripts on all these traffic generating machines to generate to generate a large number of connections varying in number of packets, duration of connection, speed, amount of data transferred and application protocol. Gigabit PickPacket was successfully tested and it captured all the packets of interest.

5.2 Performance Testing

Performance testing was carried out with a different setup than the one used for correctness testing. Test setup used for correctness testing did not really simulate the behaviour of a real network as only three machines were generating all the traffic and they were limited in speed due to various problems. Also, it was not easy to control the speed of the generated traffic with this setup. We tried to conduct the experiment on a real network where many users from different machines can be monitored. But the maximum bandwidth at the busiest link available for us was only 50 Mbps. For monitoring high and controlled speeds, we stored these packets coming at 50 Mbps to disk and later replayed them at desired speed by reading them from the disk. As one machine was not able to replay the traffic at required speed, we used multiple machines for replaying this data and evaluated the performance of Gigabit PickPacket.

Various metrics were used for evaluating the performance of Gigabit PickPacket. We used five different configuration files which covers all kinds of criteria specification. They are

1. *normal_ appl* tests all possible combinations of application level criteria specification with one criteria for each combination, but does not store any packets to disk.

2. *normal_ dump* is similar to normal_appl and it also stores around 10% packets to disk.

3. *dumpall* stores every packet to disk.

4. *heavy_ appl* tests all possible combinations of application level criteria specification with multiple (around 20) criteria for each combination, but does not store any packets to disk.

5. *heavy_ dump* includes all criteria in heavy_appl and some extra criteria to store around 10% of the read packets.

The first three configuration files put less load on Gigabit PickPacket when compared to the last two configuration files.

5.2.1 Effect of Multiple Threads/Processors

Performance of Gigabit PickPacket by varying the number of processors were carried out using a dual processor Xeon machine with hyper-threading. In all the tests, maximum bandwidth at which Gigabit PickPacket handled all the packets without any packet drop was measured. Table 5.1 shows the results obtained by varying the number of processors. The number of reading threads and processing threads, their processor binding and load on each thread are fine tuned in each case to give the maximum performance. The case of four processors was tested by using a dual processor machine with hyper-threading enabled.

Configuration File	Maximum Speed achieved with one processor (in Mbps)	Maximum Speed achieved with two processors (in Mbps)	Maximum Speed achieved with four processors (in Mbps)
normal_appl	275	300	350
normal_dump	250	275	325
dumpall	150	250	325
heavy_appl	125	200	260
heavy_dump	100	180	250

Table 5.1: Effect of number of Processors on Gigabit PickPacket's performance

It can be observed from the results that on increasing the number of processors, performance improved drastically in configurations with heavy load on Gigabit

PickPacket, while there is only a slight improvement in configurations with less load. Adding more processors did not improve performance much in configurations with less load due to two reasons. We use multiple threads to distribute application overhead among multiple processors when a single processor cannot handle the entire traffic. Here, as application load was not very high to be shared by multiple processors, we did not see much improvement in performance. The second reason is that Uniprocessor Linux kernel performs better than SMP Linux kernel in kernel level packet handling, as contention resolution in latter kernel is very costly. In configurations with heavy load, there was enough load to be shared by multiple processors and the improvement achieved due to concurrency at the application level was much more than the overhead at kernel level. When we increased number of processors from two to four, performance did not double as hyper-threading does not double the performance of processors, but only improves it by around 30% to 50%.

5.2.2 Effect of Optimizations

Table 5.2 shows the results obtained with multiple threads for the same configuration files and traffic patterns used in 5.1 but with optimized Gigabit PickPacket. NAPI for device polling, mmap to save a packet copy from kernel space to user space and a kernel module to prevent TCP/IP stack processing for each packet are the optimizations used here.

Configuration File	Maximum Speed achieved after optimizations with 1 processor (in Mbps)	Maximum Speed achieved after optimizations with 2 processors (in Mbps)	Maximum Speed achieved after optimizations with 4 processors (in Mbps)
normal_appl	325	325	400
normal_dump	300	300	375
dumpall	175	250	350
heavy_appl	125	210	275
heavy_dump	125	200	260

Table 5.2: Effect of Optimizations on Gigabit PickPacket's performance

NAPI reduces the kernel overhead in handling interrupts by polling for interrupts at regular intervals rather than device sending an interrupt after receiving packets. When a large number of packets have been received, multiple packets can be handled in a single poll in an efficient manner. Mmap saves a copy of packet from user space to memory space for each packet reaching the application layer. By preventing a copy from kernel space to user space for all the packets, kernel level packet handling overhead is reduced a lot. As the sniffed packets are not destined to reach the TCP/IP stack of the machine running the sniffer, we can safely discard them before they reach this level. This further reduces the load on kernel level packet handling. As all the optimizations result in reducing the kernel overhead rather than application overhead, performance improvement is more in configurations with less application load. In configurations with heavy load, the ratio of application overhead to kernel overhead in handling a packet is so high that the effect of optimizations does not significantly improve the overall performance.

5.2.3 Effect of Traffic Patterns

We tested Gigabit PickPacket with three different kinds of traffic patterns to find the effect of traffic pattern on performance. *d25, d50* and *d100* are the three traffic files used in this experiment. 25% of the packets in d25 belongs to the application protocols being monitored by Gigabit PickPacket, thus reach our sniffer without being filtered by the in-kernel BPF filter. Similarly d50 and d100 contains 50% and 100% packets respectively that reach the application level. Table 5.3 shows the results obtained with different traffic patterns with heavy_dump as configuration file.

It can be observed from the table that lesser the load on Gigabit PickPacket, better the performance. In d25, as 75% of the total packets are discarded at the kernel level, it gave the best possible results. As we used heavy_dump as the configuration file, packet handling time at the application layer was very high. Thus the difference in performance between different traffic patterns is quite high. A simple configuration file like normal_appl will not show much difference in performance between different traffic files. In general, we will most often see the d50 traffic pattern.

Traffic Pattern file	Maximum Speed achieved with 1 processor (in Mbps)	Maximum Speed achieved with 2 processors (in Mbps)	Maximum Speed achieved with 4 processors (in Mbps)
d25	225	300	425
d50	125	200	260
d100	80	140	180

Table 5.3: Effect of traffic pattern on Gigabit PickPacket's performance

Thus d50 is used in all the remaining tests.

5.2.4 Performance of Hub Based Approach

Performance of Gigabit PickPacket with hub based approach was measured using Xeon 2.0 GHz machine running Uniprocessor Linux 2.4.20-8. A single machine was used for this experiment. By varying the hash index from 1 to 16, load on this machine was varied from 1 to 1/16 of the total load. By scaling one machine's performance upto 16, performance results for upto 16 machines were calculated. Table 5.4 shows the results.

Configuration File	Maximum Speed achieved with 1 machine (in Mbps)	Maximum Speed achieved with 2 machines (in Mbps)	Maximum Speed achieved with 4 machines (in Mbps)	Maximum Speed achieved with 8 machines (in Mbps)	Maximum Speed achieved with 16 machines (in Mbps)
normal_appl	325	450	575	650	650
normal_dump	300	375	450	575	600
dumpall	175	500	600	650	650
heavy_appl	125	225	400	500	600
heavy_dump	125	225	400	500	600

Table 5.4: Effect of number of machines on hub based approach's performance

Multiple machines in hub based approach performed better than equal number of

processors in multithreaded approach as the kernel overhead in resolving contention is not present here. After the number of machines crossed certain limit, there is a very limited or no performance gain. In hub based approach, multiple machines share the application load but kernel load for handling all the packets is present in all the machines. Once we reach enough number of machines for sharing the application load, it is the kernel overhead that prevents us from achieving better speeds. Thus there is no performance improvement after certain level.

5.2.5 Effect of Packet Sizes

Even with a mixed approach of multithreading and hub based approach, we could not handle 1Gbps speed traffic constantly for all traffic patterns. We found that kernel limitation in handling small packets is the reason for this behaviour. We used a simple packet counting sniffer and evaluated the speed at which it started dropping the packets for different packet sizes. Table 5.5 shows the results obtained.

Packet Size	Maximum Speed achieved with packet counting sniffer (in Mbps)	Maximum Speed achieved with Gigabit PickPacket (in Mbps)
64	200	150
500	950	950
1500	1000	1000

Table 5.5: Effect of Packet size on sniffer's performance

Table shows that even an optimized packet counter cannot handle traffic speed above 200 Mbps when packet size is 64 bytes. Reference [3] discusses similar results for small sized packets. With small sized packets, number of packets received at 1Gbps speed is too high to be handled by the kernel. As kernel handles packets of different sizes in a similar manner, kernel level packet handling overhead is much higher in case of small packets. At the hardware level, individual bits are handled as signals and thus irrespective of packet sizes, hardware can handle 1Gbps traffic.

Thus we need to use a hardware splitter as described in the splitter based approach for monitoring more than 1 Gbps speed with small packets.

5.2.6 Performance of Splitter Based Approach

Due to the unavailability of a hardware splitter, we obtained the results of the Splitter approach by extrapolating the results of Gigabit PickPacket. Performance results of four processor case were extrapolated to obtain the results in Table 5.6.

Configuration File	Maximum Speed achieved with 1 machine (in Mbps)	Maximum Speed achieved with 2 machines (in Mbps)	Maximum Speed achieved with 4 machines (in Mbps)
normal_appl	400	800	1600
normal_dump	375	750	1500
dumpall	350	700	1400
heavy_appl	275	550	1100
heavy_dump	260	520	1040

Table 5.6: Effect of number of machines on splitter based approach's performance

Assuming that the hardware splitter divides the load among machines running Gigabit PickPacket in a uniform manner and it works fine at gigabit speeds without dropping any packets, we can obtain the results shown in the Table. As load balancing switches handling Gigabit speed are available in the market, these performance results can be obtained using them.

Chapter 6

Conclusions

This report discusses the filtering of packets flowing across the network based on complex criteria involving application level protocols SMTP, FTP, HTTP, Telnet and RADIUS instantiated connections at very high speeds by a network monitoring tool called Gigabit PickPacket. Various approaches that improve the performance of the monitoring tool by sharing the application processing load among multiprocessor machines and cluster of machines have been discussed. Several kernel and application level optimizations for enhancing the network monitoring tool are also discussed. Digital signature support has been added for protecting the integrity of files that contain packets stored by the filter.

Several experiments have been conducted for evaluating the performance of Gigabit PickPacket based on various metrics such as complexity of the criteria specified, traffic patterns and packet sizes. Results show that Gigabit PickPacket can monitor upto 1 Gbps traffic under very complex criteria specification for large packet sizes. In case of packets with small size, Gigabit PickPacket needs hardware splitter support for handling 1 Gbps traffic due to inherent limitation of Linux kernel in handling small packets at that speed. When complex filtering criteria are specified, support of multiprocessors and/or multiple machines can be effectively used to share the application processing overhead.

6.1 Further Work

We observed that Gigabit PickPacket could not monitor 1Gbps traffic for small sized packets without using special hardware support like splitter. One possible way to solve this problem is to use a hash function at the Network Interface Card and discard packets at the hardware level in the hub based approach thus relieving kernel from handling millions of small packets. Cost-benefit ratio of hardware splitter approach and NIC level hash function needs to be compared. Gigabit PickPacket currently does not support PASV FTP and RADIUS protocols in splitter based approach. It can be extended to support these protocols. One interesting research work would be to look at the limitations of Linux kernel in handling small packets at high speeds and propose an optimized solution to this problem.

References

[1] S. Prashant Aditya. "Pickpacket: Design and Implementation of the HTTP postprocessor and MIME parser-decoder", Dec 2002. BTP, Department of Computer Science and Engineering, IIT Kanpur, http://www.cse.iitk.ac.in/research/btp2003/98316.html.

[2] Loris Degioanni, Fulvio Risso, and Piero Viano. "Windump". http://netgroup-serv.polito.it/windump.

[3] Luca Deri. "Improving Passive Packet Capture: Beyond Device Polling". http://luca.ntop.org/Ring.pdf.

[4] Gerald Combs et al. "Ethereal". Available at http://www.ethereal.com.

[5] Robert Graham. "carnivore faq". http://www.robertgraham.com/pubs/carnivore-faq.html.

[6] "How Carnivore Works". http://www.howstuffworks.com/carnivore.htm.

[7] Van Jacobson, Craig Leres, and Steven McCanne. "tcpdump : A Network Monitoring and Packet Capturing Tool". Available via anonymous FTP from ftp://ftp.ee.lbl.gov and www.tcpdump.org.

[8] Sanjay Kumar Jain. "Implementation of RADIUS Support in Pickpacket". Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, Apr 2003. http://www.cse.iitk.ac.in/research/mtech2001/Y111122.html.

[9] Neeraj Kapoor. "Design and Implementation of a Network Monitoring Tool". Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, Apr 2002. http://www.cse.iitk.ac.in/research/mtech2000/Y011111.html.

[10] Steve McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture". In *Proceedings of USENIX Winter Conference*, pages 259–269, San Diego, California, Jan 1993.

[11] "nprobe nflow". http://www.ntop.org/nFlow/.

[12] Brajesh Pande. "Design and Implementation of a Network Monitoring Tool". Master's thesis, Department of Computer Science and Engineering, IIT Kanpur, Sep 2002. http://www.cse.iitk.ac.in/research/mtech2000/Y011104.html.

[13] Boyer R. and J Moore. "A fast string searching algorithm". In *Comm. ACM 20*, pages 762–772, 1977.

[14] Stephen P. Smith, Henry Perrit Jr., Harold Krent, Stephen Mencik, J. Allen Crider, Mengfen Shyong, and Larry L. Reynolds. "Independent Technical Review of the Carnivore System". Technical report, IIT Research Institute, Nov 2000. http://www.usdoj.gov/jmd/publications/carniv_entry.htm.

[15] "Sniffer Portable by Network Associates". http://www.networkassociates.com/.

[16] "TopLayer IDS load balancers". http://www.toplayer.com.

[17] "Unispeed Netlogger". http://www.unispeed.com/.

[18] Jacobson V., Leres C., and McCanne S. *"pcap - Packet Capture Library"*, 2001. Unix man page.

Appendix A

A Sample Configuration File

```
#This is a sample configuration file
#Be very careful if you edit a configuration file manually
# The syntax should be preserved
# A hash(#) is used for comments
# This file has several sections
#Sections start and end with tags similar to HTML.
#Tags within sections can start and end subsections or can be tag-value pairs.
#All the tags that are recognized appear in this file.
# First Section spcifies the sizes and names of the dump files
# The Second Section specifies the source and destination IP ranges
# the source and destination ports, the protocol and the application
# that should handle these IPs and ports
# The third sections specifies the number  of connections to open simultaneously
# for some applications
# The fourth section specifies the thread specific information and hash values.
# The next sections describe the application specific
# input criteria.
# This file has a fixed format Careful!!

#**************First Section*****************************
<Output_File_Manager_Settings>
   <Default_Output_File_manager_Settings>
#number of specified files
       Num_Of_Files=2
#the full file name relative/absolute will do
       File_Path=dump1.dump
#the file size in MB
       File_Size=12
```

```
       File_Path=dump2.dump
#the 0 file size means that file can be of max available size
#only the last file can have File_Size=0.
       File_Size=0
   </Default_Output_File_manager_Settings>
</Output_File_Manager_Settings>
#**************End First Section************************

#**************Second Section*****************************
# The basic criteria here are for the Device and
# SrcIP1:SrcIP2:DestIP1:DestIP2:SrcP1:SrcP2:DestP1:DestP2:ProtoA:App
# Should be read as For the range of sorce IP from SrcIP1 to SrcIP2
#                   For associated ports from SrcP1 to SrcP2
# and For the range of desitnation IP from DestIP1 to DestIP2
#                   For associated ports from DestP1 to DestP2
#               and FOR Protocol ProtoA
#               monitor connections according to Application App
# Protocols can be UDP or TCP
# Applications for TCP are
#     SMTP, FTP, HTTP, TELNET, RADIUS, TEXT, DUMP_FULL, DUMP_PEN
# Applications for UDP are
#     DUMP_FULL, DUMP_PEN
# No further specs are required for DUMP kind of applications.
# Do not mix too many applications for clarity
# Take care that IPs Ports and applications do not conflict
# Important: Some old NAS/RAS sends packets assuming RADIUS Auth Server port
# as 1645 and Accounting Server port as 1646. So for this type of RAS/NAS we
# need to change server port
# in configuration file as mentioned in next two lines.
# Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:1645-1645:UDP:RADIUS
# Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:1646-1646:UDP:RADIUS
<Basic_Criteria>
    DEVICE=eth0
Num_Of_Criteria=10
   Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:25-25:TCP:SMTP
   Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:20-20:TCP:FTP
   Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:21-21:TCP:FTP
   Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:23-23:TCP:TELNET
   Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:80-80:TCP:HTTP
   Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:143-143:TCP:TEXT
   Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:1024-65535:TCP:DUMP_FULL
```

```
   Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:1024-65535:UDP:DUMP_FULL
   Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:1812-1812:UDP:RADIUS
   Criteria=0.0.0.0-0.0.0.0:0.0.0.0-0.0.0.0:1024-65535:1813-1813:UDP:RADIUS
</Basic_Criteria>
#**********End Second Section*****************************

#**************Third Section*****************************
# Has tunable number of connections that should be monitored
# by some applications of interest SIMULTANEOUSLY
<NUM_CONNECTIONS>
   NUM_CONNECTIONS=5
   NUM_SMTP_CONNECTIONS=500
   NUM_FTP_CONNECTIONS=500
   NUM_HTTP_CONNECTIONS=500
   NUM_TELNET_CONNECTIONS=500
   NUM_RADIUS_CONNECTIONS=500
</NUM_CONNECTIONS>
#**********End Third Section*****************************
#**************Fourth Section*****************************
# Information regarding hash values and hash range for this
# machine are present here. Each subsection contains information
# about one thread. As many subsections as the number of
# threads to be created are present in this section.
<THREAD_INFO>
   HASH_INDEX=8
   NUM_THREADS=2
   HASH_START_VALUE=0
   HASH_END_VALUE=3
   <Thread>
      Type=Processing
      Processor=0
      Load=1
      Pending_Queue_Length=1000
   </Thread>
   <Thread>
      Type=Reading
      Processor=1
      Load=3
      Pending_Queue_Length=1000
   </Thread>
</THREAD_INFO>
```

```
#**********End Fourth Section**************************

#**************Application Specific Specifications******
#If there are RADIUS Specific criteria then those criteria comes first in this file
#**************RADIUS Specifications******
<RADIUS_Configuration>
   Num_Of_Criteria=3
   Criteria=skjaincs:no:0.0.0.0-0.0.0.0:1024-65535:1-65535:TCP:DUMP_FULL
   Criteria=vijayg:no:0.0.0.0-0.0.0.0:1024-65535:25-25:TCP:SMTP
   Criteria=vijayg:no:0.0.0.0-0.0.0.0:1024-65535:23-23:TCP:TELNET
</RADIUS_Configuration>
#**************SMTP Specifications******
<SMTP_Configuration>
   <SMTP_Criteria>
       NUM_of_Criteria=2
       <Search_Email_ID>
           Num_of_email_id=1
           Case-Sensitive=yes
           E-mail_ID=skjaincs@cse.iitk.ac.in
       </Search_Email_ID>
       <Search_Text_Strings>
           Num_of_Strings=1
           Case-Sensitive=yes
           String=book
       </Search_Text_Strings>
       <Search_Email_ID>
           Num_of_email_id=2
           Case-Sensitive=yes
           E-mail_ID=skjaincs@iitk.ac.in
           E-mail_ID=brajesh@hotmail.com
       </Search_Email_ID>
       <Search_Text_Strings>
           Num_of_Strings=0
       </Search_Text_Strings>
   </SMTP_Criteria>
   Num_of_Stored_Packets=750
   Mode_Of_Operation=full
</SMTP_Configuration>
#**********END SMTP Specifications******

#**********FTP Specifications******
```

```
<FTP_Configuration>
   <FTP_Criteria>
       NUM_of_Criteria=1
       <Usernames>
           Num_Of_Usernames=2
           Case-Sensitive=no
           Username=ankanand
           Username=nmangal
       </Usernames>
       <Filenames>
           Num_Of_Filenames=1
           Case-Sensitive=no
           Filename=test.txt
       </Filenames>
       <Search_Text_Strings>
           Num_Of_Strings=1
           Case-Sensitive=yes
           String=book secret
       </Search_Text_Strings>
   </FTP_Criteria>
   Num_of_Stored_Packets=750
   Monitor_FTP_Data=yes
   Mode_of_Operation=full
</FTP_Configuration>
#**********END FTP Specifications******

#*************HTTP Specifications******
<HTTP_Configuration>
   <HTTP_Criteria>
       NUM_of_Criteria=1
      <Host>
           Num_Of_Hosts=1
           Case-Sensitive=no
           HOST=http://www.rediff.com
       </Host>
       <Path>
           Num_Of_Paths=1
           Case-Sensitive=yes
           PATH=/cricket
       </Path>
       <Search_Text_Strings>
```

```
            Num_of_Strings=1
            Case-Sensitive=no
            String=neutral venu
        </Search_Text_Strings>
    </HTTP_Criteria>
    <Port_List>
        Num_of_Ports=1
        HTTP_Server_Port=80
    </Port_List>
    Num_of_Stored_Packets=750
    Mode_Of_Operation=full
</HTTP_Configuration>
#*********END HTTP Specifications******

#*********TELNET Specifications******
<TELNET_Configuration>
    <Usernames>
        Num_of_Usernames=1
        Case-Sensitive=yes
        Username=ankanand
    </Usernames>
    Mode_Of_Operation=full
</TELNET_Configuration>
#*****END TELNET Specifications******

#*********TEXT SEARCH Specifications******
#These have to be added manually

<TEXT_Configuration>
    <Search_Text_Strings>
        Num_of_Strings=1
        Case-Sensitive=no
        String=timesofindia
    </Search_Text_Strings>
    Mode_Of_Operation=pen
</TEXT_Configuration>
#*****END TEXT SEARCH Specifications******

#**********End Application Specific Specifications****
```