國 立 交 通 大 學

資訊科學系

碩 士 論 文

一個獨特的閘道器架構用來管理使用動態連接埠的點對點連線

A Novel Gateway Architecture for Managing Dynamic Port Peer-to-Peer Traffic

研　究　生：蔡孟甫

指導教授：林盈達　教授

中 華 民 國　九 十 四　年 六 月

一個獨特的閘道器架構用來管理使用動態連接埠的點對點連線

A Novel Gateway Architecture for Managing Dynamic Port Peer-to-Peer traffic

研 究 生： 蔡孟甫	Student : Meng-Fu Tsai
指導教授： 林盈達	Advisor : Ying-Dar Lin

國 立 交 通 大 學

資 訊 科 學 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer and Information Science

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer and Information Science

June 2005

HsinChu, Taiwan, Republic of China

中華民國九十四年六月

一個獨特的閘道器架構用來管理
使用動態連接埠的點對點連線

學生：蔡孟甫　　　　　指導教授：林盈達

國立交通大學資訊科學系

摘要

傳統閘道器架構無法管理使用動態連接埠的點對點連線，我們提出一個整合新架構來管理下面項目：1. 辨識連線屬於那個應用程式 2. 過濾不被允許的應用程式 3. 對傳送的檔案進行掃毒 4. 過濾並且監控聊天訊息或是傳送的檔案 5. 對特定應用程式進行頻寬管理。此架構在核心進行連線辨識並在使用者層進行複雜的內容管理。在核心中有兩個封包佇列，一個多執行緒的使用者層程式與核心一同同步操縱這兩個封包佇列中的封包。使用者層的程式使用這兩個封包佇列來解決封包順序錯誤的問題還有線頭阻擋的問題。外部測試顯示這個架構的吞吐量能達到 84.83 Mb/s，但是如果啟動掃毒功能，吞吐量馬上降到 20.52 Mb/s。內部測試顯示掃毒花的時間是其他步驟的 200 至 800 倍。而與傳統利用連接埠來重導連線的代理伺服器相比，連線辨認與導引影響吞吐量大約 40 Mb/s。

關鍵字：點對點、順序錯亂、*線頭*、*內容過慮*、*代理伺服器*

A Novel Gateway Architecture for Managing Dynamic Port Peer-to-Peer Traffic

Student: Meng-Fu Tsai Advisor: Dr. Ying-Dar Lin

Department of Computer and Information Science

Nation Chiao Tung University

Abstract

Conventional port-redirect proxy architecture can not manage peer-to-peer (P2P) traffic which might run over dynamic ports instead of fixed well-known ports. We propose a novel gateway architecture for five management objectives: (1) connection classification of P2P applications, (2) filtering undesirable P2P traffic, (3) virus scanning on P2P shared files, (4) filtering and auditing of chatting messages and transferred files and (5) bandwidth control of the P2P traffic. This architecture performs connection classification and complex content management in the kernel and user space, respectively. The packets of identified connections are queued in the kernel. There are two packet queues in the kernel. A multi-threaded proxy program cooperates with the kernel to manipulate packets in theses two packet queues synchronously to solve the packet out-of-order problem and the head-of-line blocking problem. The external benchmarking reveals that the throughput of this architecture can achieve 84.83 Mb/s. But if enable the virus scanning function, the throughput decreases to 20.52 Mb/s. The internal benchmarking reveals that the time spent on virus scanning is 200 ~ 800 times than other steps. Comparing with port-redirect proxy, the impact of connection classification and redirection is about 40Mb/s.

Keywords: peer-to-peer, out-of-order, head-of-line, content filtering, proxy

Contents

List of Figures

List of Tables

Chapter 1 Introduction

Over the last few years, peer-to-peer (P2P) file sharing has grown astonishingly in the Internet. System administrators used to manage Internet traffic by identifying it according to *fixed* well-known port numbers. The management includes blocking traffic of specific applications or redirecting the connections to the proxy that performs various kinds of content filtering such as virus scanning. Nonetheless, the identification for P2P traffic is non-trivial because most P2P applications may use *dynamic* ports, i.e. dynamically selected ports rather than fixed well-known ports.

The P2P traffic can be identified either by examining packet payloads [1] or analyzing the connection pattern at the transport layer [2] . Both approaches demand a connection to be established between two peers before the identification. But for P2P management, how to redirect the connection from the kernel to an application proxy to perform content filtering *after* the connection has been established is rarely addressed in both research and industry fields to the best of our knowledge.

This work designs a novel gateway architecture to manage P2P traffic. The management objectives in the architecture cover (1) connection classification, or identification of P2P applications, (2) filtering undesirable P2P applications, (3) virus scanning for P2P shared files, (4) filtering and auditing of chatting messages and transferred files and (5) bandwidth control of the P2P traffic.

The L7-filter [3] serves as a connection classifier that identifies P2P applications according to the signatures in the application-layer messages. The identification is executed in the kernel space because it is a simple signature matching from the first few bytes. Objectives (2) and (5) follow immediately by referring to the identification results. However, Objectives (3) and (4) typically involve more complex content processing and filtering from data assembled from packets; thus they are better executed in the user space. The latter requires connection *redirection* from the kernel to the user space. This work designs a new mechanism to address this problem in the software architecture.

A connection is marked after being classified. Only the packets of the marked connection are queued in the kernel. The queued packets are then *duplicated* to the user space, where the proxy program performs the necessary content filtering to decide whether to pass or drop the packets in the kernel queue. Because the proxy receives raw packets from the kernel, there might be packet *out-of-order* problem and the proxy should perform TCP reassembly. Since the proxy handles queued packets sequentially, the time-consuming content filtering may causes *head-of-line* blocking in the kernel queue, where the packets of other connections are queued behind the packet being examined in the proxy. This work thus proposes a mechanism which uses two packet queues to handle the foregoing situations. The entire mechanism of this architecture is called *P2P Proxy Mechanism* and the proxy program is called *P2P Proxy*.

P2P Proxy Mechanism is implemented in Linux kernel 2.6.8. A modified queue handler, *ip_queue,* queues the packets with two packet queues and then the library *libipq* [4] are modified to let the proxy *manipulate* packets in theses two kernel queues. The proxy is multi-threaded. The main thread handles packet arrivals, and the others handle specific application protocols and perform content filtering.

In this work, we want to answer the following questions: (1) What is the overhead of P2P Proxy Mechanism compared with that of the simple port-redirect architecture? (2) What is the main bottleneck of this system? (3) What is the difference of performance between P2P Proxy Mechanism and virus scanning?

The rest of this work is organized as follows. Chapter 2 surveys present P2P applications and lists our management objects. Chapter 3 presents our ideas and system architecture. The implementation details, including the selected packages and thread implementation details are illustrated in Chapter 4. Chapter 5 discusses the performance of our system. We conclude the study in Chapter 6.

Chapter 2 Survey and Problem Statement

2.1 Related Works

Research about P2P traffic mostly emphasizes on connection classification to date. Lots of them only consider the traffic on fixed port [5] [6] [7] . Recent works try to identify P2P traffic which uses dynamic ports. Two major approaches are examining the bit strings in the packet payloads [1] and analyzing the P2P flows at the transport layer according to the connection patterns of P2P Network [2] . Both demand a connection to be established between two peers before the identification. The former can identify the P2P protocol by matching its signatures, but it can do that only if the signatures are known. The latter can identify unknown P2P traffic, but it cannot decide immediately whether this connection is some P2P since it needs the statistics of flows for a while. This method cannot point out exactly what application the connection belongs to.

Some open source packages, such as L7-filter [3] and IPP2P [8] , are also developed to identify P2P traffic. They are both classifiers that inspect the packet payload in the Linux Netfilter [9] subsystem. The L7-filter uses Netfilter's connection-tracking module and only checks the first *eight* packets for the application data when a connection is established. If the application data matches the signature, it marks the entire connection as identified by the connection-tracking module. Wile IPP2P checks every packet, this is because it does not adopt connection-tracking module. The other difference is that the signatures of IPP2P are hard-coded but L7-filter can load signatures from files. Therefore, inspecting fewer packets and dynamically loading signatures gives L7-filter higher performance and better scalability than P2PADM. This work presents a complete architecture integrated with the L7 filter for various kinds of P2P management objectives.

For proxy architecture research, there are some research for improving proxy performance, little research has been done on topic development in improvement of TCP splicing for application proxy performance with kernel support[10] [11] . There is also study in reducing overheads to minimize system costs [12]

2.2 Problem Statement

After P2P connection classification, blocking undesirable applications and bandwidth control on specific applications can be enforced, which all are done within the kernel. However, there is still no content management processing for the P2P traffic. The difficulty is how to redirect the connection from the kernel to an application proxy to perform content filtering after the connection has been established and classified in the kernel. Way to solve this problem and what kinds of management objectives can be reached are described in this work.

2.3 P2P and IM Applications Overview

Popular P2P applications include eDonkey[13] , BitTorrent [14] , FastTrack[15] Gnutella[16] , etc. Besides, file transfer in the Instant Messenger (IM), say MSN[17] , also works in the P2P mode. Most P2P applications use dynamic ports to circumvent filtering firewalls. Table 1 summarizes the characteristics of these applications.

These P2P applications have two modes when transferring files. One is sequential transfer, which means a peer receives a file sequentially from another peer. The other is *segmented* transfer, which means that the segments of a file can be received out-of-order. System administrators may want to scan the transferred file for viruses and record what files are transferred. The data cannot be segmented out-of-order or encrypted in order to perform virus-scanning or recording. According to Table 1, these two actions can only be done for FastTrack, MSNFTP and Gnutella. If the file name is visible, filtering the file name which contains specific keyword is possible. Enterprises may not want employees to leak out confidential information by a chatting system like the IM. Therefore, filtering the sensitive keywords or recording the message is needed. Table 2 lists the possible management objectives for each application protocol. The proposed architecture intends to implement these management objectives.

Table 1: The characteristics of P2P and IM applications.

Application Protocol	FastTrack	eDonkey	BitTorrent	Gnutella	MSN	MSNFTP*
Is file transfer sequential?	Yes	No	No	Yes	N/A	Yes
Protocol message encryption	Yes	No	No	No	No	No
Data transfer encryption	No	No	No	No	No	No
Can use dynamic port?	Yes	Yes	Yes	No	No	Yes
File name visibility?	Maybe	No	Yes	Yes	Yes	No
Default ports	1214	4661-4665	6881-6889	6346-6347	1863	No default

***MSNFTP is a file transfer protocol of MSN. N/A = not available**

Table 2: Management objectives for each application protocol

Application Protocol	FastTrack	eDonkey	BitTorrent	Gnutella	MSN	MSNFTP
Connection classification	O	O	O	O	O	O
Filtering undesirable applications	O	O	O	O	O	O
Virus scanning	O	X	X	O	N/A	O
Filtering and auditing of chatting messages and transferred files	O	X	X	O	O	O
Bandwidth control	O	O	O	O	O	O

Chapter 3 System Architecture Design

3.1 Solution Ideas

Because P2P connection classification needs to examine application messages, the TCP connection between two peers must be established first. However, conventional port redirection method can not be applied to a connection already established. But since all packets between two peers must pass through our gateway, it is possible to perform content management by handling these packets. We want the kernel to get the packets and then cooperate with the proxy for further processing. This work proposes a novel architecture to enable classification as well as management on these P2P traffic. The steps can be high level described as follows: (1) Use the L7-filter to perform connection classification and marking. (2) Queue the packets in the kernel and wait the verdict from the user-space proxy. (3) The proxy handles packet classification and out-of-order packets. (4) The proxy decides the verdict according to the management objectives. (5) The proxy solves the head-of-line blocking and virus signature segmentation.

3.2 In-kernel Connection Classification and Marking

We adopt the L7-filter to perform connection classification in kernel. It collects at most the first eight packets to reassemble an application message and do signature matching. If the connection is identified by the L7-filter, it will be marked by a predefined application number. The kernel can filter the undesirable applications and do bandwidth control according to this predefined application number. Complex content filtering functions such as virus scanning should be processed in the user space. How to redirect packets to the user space is described in section 3.3. But the L7-filter needs to collect enough data to perform signature matching for connection classification. It collects the application data of first eight packets into a buffer. After successful matching, the first few packets might have already passed. These packets may contain important protocol information such as the file name or the file size. And the proxy, however, may need this information to take the corresponding action. To solve this problem, after successful matching, a specific packet is created in the kernel and the application data collected by the L7-filter will be inserted to this packet. This specific packet will only be passed to the proxy rather than being transmitted out. When the proxy gets this specific packet, it will not lose any application data in the previous packets.

3.3 In-kernel Packet Queuing and Redirect Mechanism

In kernel, two packet queues Q1 and Q2 are created to manage the P2P traffic. All packets identified by the L7-filter are queued in Q1. Those unidentified packets are just passing through or processed by the filtering firewall. Then the queued packets are copied to the user space and waits for the verdict of the proxy. The proxy processes and sets verdict for those packets queued in Q1 sequentially. The verdict from the proxy may be ACCEPT, DROP or QUEUE. ACCEPT will let the packet pass and DROP will drop the packet. If the packet can not be decided to be passed or dropped at that time, the verdict QUEUE will be set and the packet will be moved from Q1 to Q2. How to handle the packets in Q2 and move them back to Q1 will be described in later sections. Figure 1 illustrates this mechanism. How proxy makes the verdict is described in section 3.4 and 3.5.

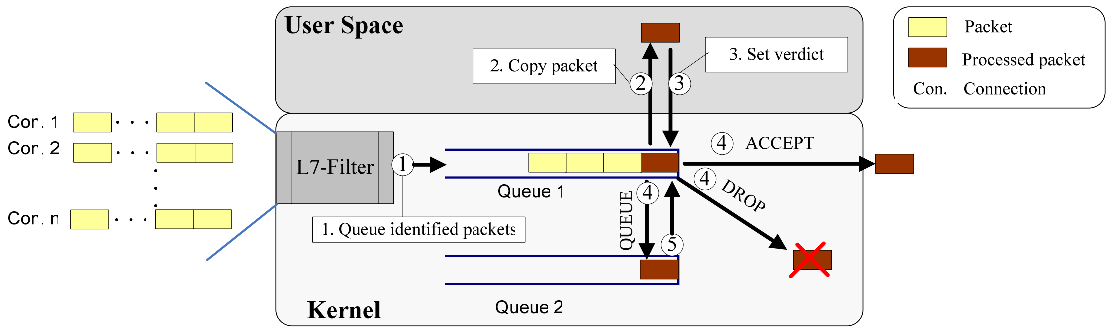


Figure 1: Packet queuing and redirect mechanism

3.4 In-daemon Packet Pre-processing

When the proxy gets the packet from Q1, three tasks must be done before handling the specific application protocol. First the packet checksum is examined. If the checksum is in error, the proxy does not process this packet and just tells the kernel to pass this packet instead of dropping this packet. This is because the connection reliability is the responsibility of two peers, not this gateway. The second is packet classification, i.e. identify what connection this packet belongs to, and the third is handling out-of-order packets. The former is needed because the kernel only uses one queue to queue the packets of all marked connections, which means that this queue contains packets of various connections. Therefore, the proxy needs to identify which connection a packet belongs to. Packet classification is performed based on the five tuples, i.e., Source IP address, source port, destination IP address, destination port and protocol identifier.

After packet classification, packets may still be out-of-order. This is because the redirected packets do not pass through the TCP stack. To handle this problem, the proxy calculates the next correct sequence number in advance and checks the sequence number of the handled packet. If the sequence number of the packet is less then the correct sequence number, this means that it is a duplicated packet and is just passed without any processing. If the sequence number of the packet is larger than the correct sequence number, this means the packet should wait until the appearances of all packets with the sequence number smaller than this one. Therefore, we do not process this packet and tell the kernel to move this packet from Q1 to Q2. If the sequence number of the packet is just correct, this packet is processed and the out-or-order packets in Q2, if any, will be moved to Q1. The packet out-of-order problem is solved in this way.

We only take care of the packet ordering problem instead of implementing the entire TCP stack. This is because the TCP reliability, again, is the responsibility of two peers. Peers should handle packet checksum error and duplicated packets. If there are lost packets, the peer should retransmit them. What we care is only the contiguous application data. Therefore, if the packet out-of-order problem is solved, then the TCP reassembly is trivial.

3.5 In-daemon Application Protocol Processing

After getting the packet with the correct sequence number, the last thing to do is to handle the application protocol. Processing which application protocol is according the packet mark number. There are two common management objectives to achieve. The first is filtering and auditing of the chatting messages and transferred file. The second is virus scanning on shared files. Other management objectives may exist, but are not addressed here. Since our management objectives mostly are related to file transfer, we observe the procedure of content processing for file transfer. In our observation, a connection has three states when transferring file: (1) Initial state: waiting for the file transfer request and response, (2) Receiving state: receiving the transferred data, (3) Processing state: The proxy performs the content filtering on the receiving data. Theses three connection states will be used in the later sections.

3.5.1 Filtering and Auditing of Chatting Messages and Transferred Files.

After the pre-processing tasks, the packet is checked based on the corresponding application protocol to examine if the chatting message or file transfer request contains the specific keywords. If a specific keyword is found, the proxy tells the kernel to drop the packet and send the RST packet to the source peer to break down the connection. Otherwise the proxy tells the kernel to pass this packet and recording the chatting message or transferred file name. If this packet is a file transfer request and there is no specific keyword in the file name, the connection is marked to the receiving state. For the packets coming later, we just record the transferred data segments into a file and do virus scanning until the file transfer is completed.

3.5.2 Virus Scanning for Shared Files

For virus scanning, a buffer is allocated for each connection. Instead of doing virus scanning for each packet, virus scanning is only performed when enough data has been collected. There are two reasons. First, a virus signature might appear across the boundary of two consecutive data segments or it could be longer than a single segment. Second, the overhead of calling the virus scanning function is huge, too many virus scans on small segments size is inefficient. Hence, when the proxy gets a packet, first it checks if the buffer is full or if this is the last data segment of the transferred file. If neither one happens, the data segment in the packet payload is saved into the buffer. Then the proxy tells the kernel to pass this packet. If any foregoing situation happens, the proxy performs virus scanning on the buffer. If the virus is found, the proxy tells the kernel to drop this packet and send the RST packet to the peer to break down the connection. Dropping this packet can ruin the whole file. If no virus found, we clean the buffer and also let the packet pass.

The above method has two problems described as follows: the first is the head-of-line blocking and the other is the segmented virus signature. Two mechanisms are proposed to avoid these problems.

Head-of-line blocking

The head-of-line blocking happens because virus scanning is time-consuming. Other packets queued in Q1 can not be handled until virus scanning on its buffer is finished. This will limit the throughput of the entire system. Figure 2(a) shows the packet processing time of this situation. To solve this problem, when virus scanning is needed, the connection is marked to the *processing* state and another thread is called to perform virus scanning. Then the proxy tells the kernel to move subsequent packet arrivals of this connection from Q1 to Q2. Therefore we can immediately handle the packets of other connections in Q1. When virus scanning is finished, if virus is found, all queued packets of this connection in Q2 will be dropped, otherwise theses queued packets are moved *back* from Q2 to Q1. By this mechanism, the head-of-line blocking problem is avoided. Figure 2 (b) shows the packet processing time without handling head-of-ling blocking. Obviously, (b) achieves better concurrency between connections.

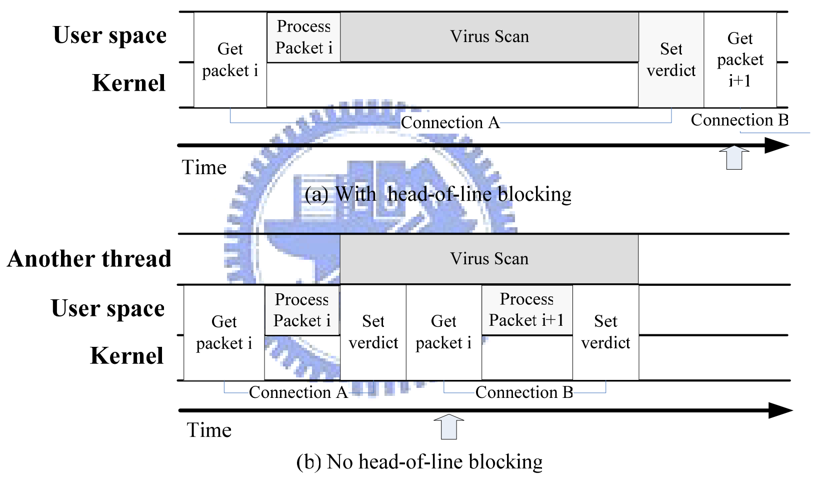


Figure 2: Head-of-line blocking problem

Segmented virus signature

Because each time only a block of the entire transferred file is scanned, it is possible that a virus signature may be segmented into two data blocks. To solve this problem, when the virus scanning finishes, the tail data of length S will be kept in the buffer instead of cleaning the entire buffer, where S is the max length of virus signatures. The subsequent data segments are appended to the buffer. By this mechanism, we can still detect segmented virus signatures.

3.6 Final Proxy Architecture Design

We integrate all the above the ideas into a complete proxy. The entire mechanism is called *P2P Proxy Mechanism*. In summary, the kernel queues the packets of the classified connections. In the proxy, there is a main thread to get packets from Q1 in the kernel and perform the pre-processing tasks. We must keep in mind that Q1 contains the packets of various connections which we want to manage and the proxy uses the application number marked on the packet to identify which application this packets belongs to. Then the main thread calls a specific application thread to handle the tasks related to the application protocol according to the application number mentioned previously. Each application thread is responsible for a specific connection. Figure 3 illustrates the entire flow chart of the main thread. After performing the pre-processing tasks described in section 3.4, the main thread checks the connection state. If the connection state is in the processing state, it needs to solve the head-of-line blocking problem. Otherwise it calls the specific application thread to handle the packet.

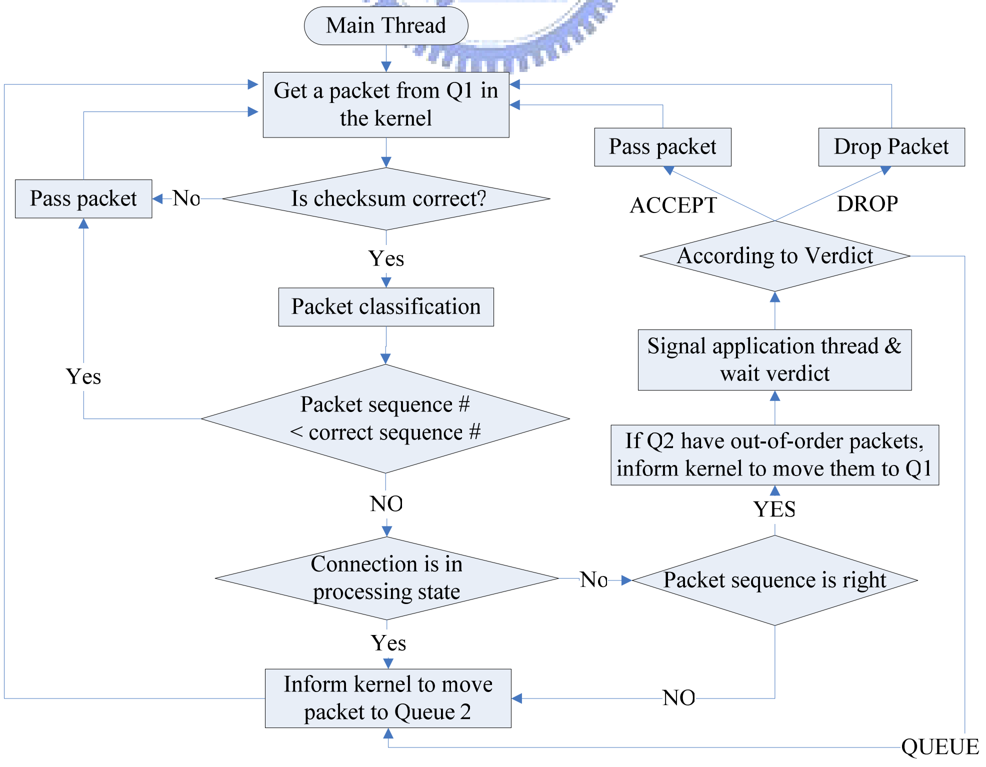
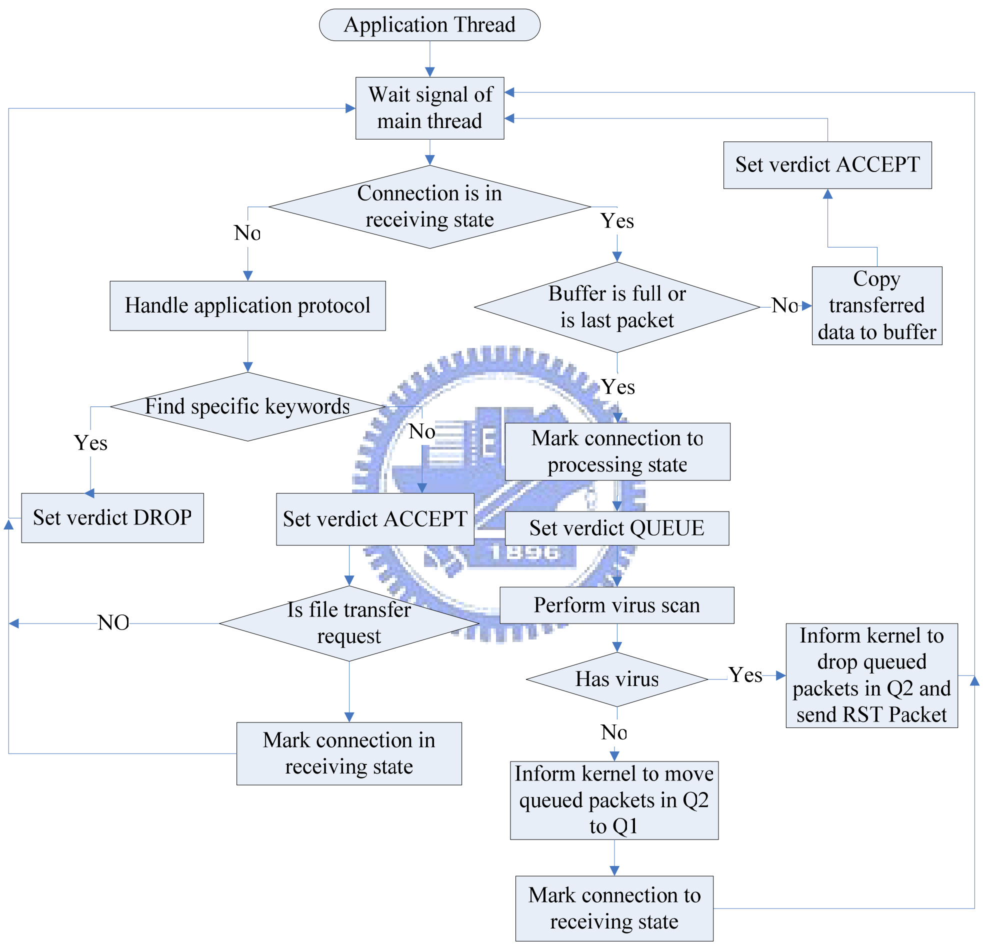

Main Thread
Get a packet from Q1 in the kernel
Pass packet
Drop Packet
Pass packet
No
Is checksum correct?
ACCEPT
DROP
According to Verdict
Yes
Packet classification
Signal application thread & wait verdict
Yes
Packet sequence # < correct sequence #
If Q2 have out-of-order packets, inform kernel to move them to Q1
NO
YES
Connection is in processing state
No
Packet sequence is right
Yes
Inform kernel to move packet to Queue 2
NO
QUEUE

Figure 3: The flow chart of main thread

Figure 4 illustrates the entire flow chart of an application thread. The application thread handles the application protocol and decides to pass or drop the packet. If it needs to perform time-consuming content filtering, such as virus scanning, it should mark this connection in processing state and set the verdict to QUEUE. Then the main thread will star to process the next packet. This will avoid the head-of-line blocking. How we implement this system is described in Chapter 4.



Application Thread
Wait signal of main thread
Set verdict ACCEPT
Connection is in receiving state
No
Yes
Handle application protocol
Buffer is full or is last packet
No
Copy transferred data to buffer
Find specific keywords
Yes
Yes
No
Mark connection to processing state
Set verdict DROP
Set verdict ACCEPT
Set verdict QUEUE
NO
Is file transfer request
Perform virus scan
Inform kernel to drop queued packets in Q2 and send RST Packet
Has virus
Yes
No
Mark connection in receiving state
Inform kernel to move queued packets in Q2 to Q1
Mark connection to receiving state

Figure 4: The flow chart of an application thread.

Table 3 lists the differences between the architectures of a conventional proxy and our P2P proxy. Four major differences are (1) the connection classification method, (2) the traffic redirection mechanism, (3) whether proxy handles packet reassembly itself and (4)

Table 3: The difference in proxy architecture

	Conventional proxy	**P2P Proxy**
Connection classification	By port number	By application signature
Traffic redirection	Two connections set up between the proxy program and either end of peer	Only one connection between the two peers. The proxy interferes the connection and captures the packets to the user space for verdict.
Whether to handle reassembly itself	Reassembled from the TCP/IP stack	Reassembly itself, and then analyze the reassembled data
Synchronization between proxy and the kernel	None	Use two packet queues in the kernel to synchronize.

Chapter 4 System Implementation

4.1 Selected OS and Packages

P2P Proxy Mechanism is implemented on Linux 2.6.5 and the proxy program is called P2P Proxy. The reason of selecting Linux as our platform is because that L7-filter depends on Netfilter that can only run on Linux. The selected packages, libraries and module are listed in Table 3. How we patch ip_queue and libipq are described in section 4.2.2. And how we implement P2P Proxy is described in section 4.2.3.

Table 4: Selected packages and modules

Name	**Type**	**Description**	**Pacthes**
L7-filter	Match module of Netfiler	Perform connection classification	After signature matching, send a specific packet to user space program
ip_queue	Kernel module of Linux	Packet queuing handler	Another packet queue is added
Libipq	Library	Let user-space programs to get packets from the queue in in_queue and set verdict of packets	Add a function to control two packet queues in ip_queue
Libclamav	Library	Virus scanning engine of ClamAV[18]	None
TC	Package	Used for bandwidth control	None

4.2 System Implementation Details

The following sections describe the implementation details of this system. We develop our system by integrating the foregoing packages and modifying some of them.

4.2.1 Application filtering and bandwidth control

The L7-filter is used to identify the specific application traffic that we want to manage. The identified application traffic is marked by Netfilter with a predefined application number. This number is recorded in the sk_buff, which is the data structure used to represent the packet in Linux. According to this number, filtering an undesirable application can be enforced by using Netfilter to drop the packets with the specific number. Bandwidth control is also based on this application number. We configure TC to classify traffic according to this number. Each application number represents a traffic class and the connections of the same application number belong to the same class. Each class only occupies the limited bandwidth.

4.2.2 *ip_queue and libipq*

To perform more complex content filter in the user space, modifying *ip_queue* and *libipq* are needed to let the user-space proxy to communicate with the kernel. In this work, there are tight communications between the kernel and the proxy. The kernel module *ip_queue* and the library *libipq* are modified to let the proxy control packets in the kernel. The original *ip_quque* is a queue handler with only one queue to queue packets and the original *libipq* is a library for the proxy to set verdict to the packets queued in the *ip_queue* module. Another queue is added to *ip_queue* in our modification and *libipq* is also modified to control packets between these two queues. These two components are modified and the proxy uses them to control packets.

4.2.3 Multi-threaded Proxy

P2P Proxy is multi-threaded. First, this system is to run on an embedded gateway. A multi-process architecture would occupy too many system resources. Second, the internal process communication (IPC) can be reduced. This is due to the shared memory space between threads. Those shared data can be access directly without redundant IPC.

Figure 5 illustrates the complete implementation architecture. The main thread receives queued packets from Q1. Each application module uses a thread pool to handle application protocol. The main thread and all application modules share the data space. The following items are recorded in the shared data space.

1. The configuration of each application, such as which functions should be turned on and the keywords that should be filtered.
2. A connection hash table to save information of the application to which each connection belongs. The information recorded for each connection includes next sequence number, current state and buffer usage information.

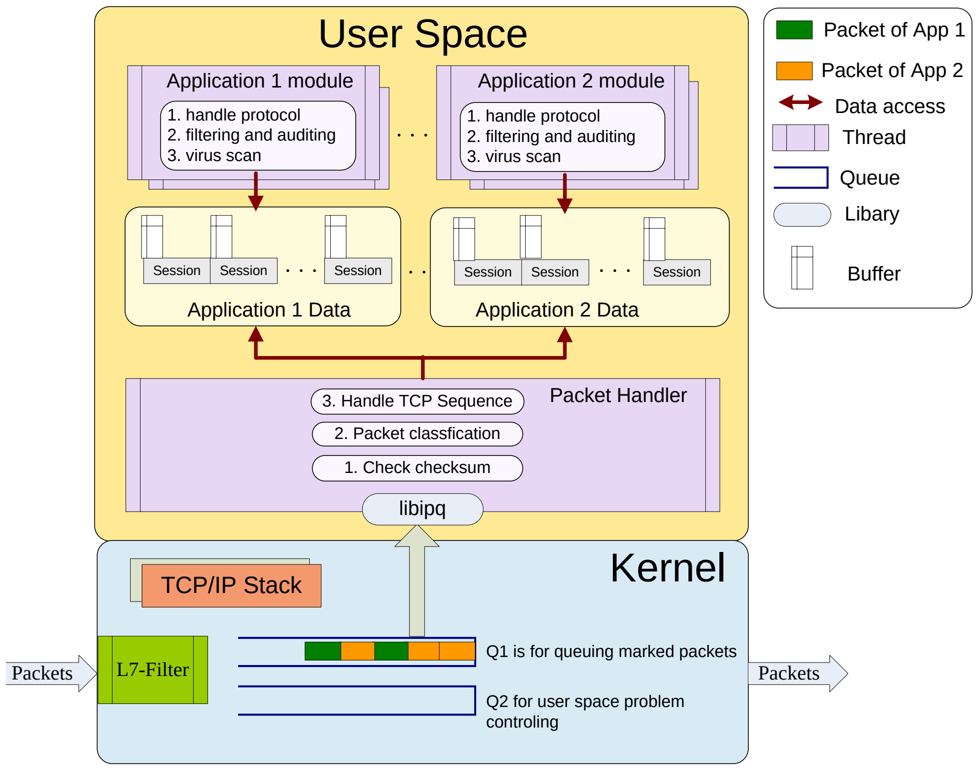


Figure 5: System implementation architecture

When the main thread uses *libipq* to get the packets identified by L7-filter, it performs the packet preprocessing tasks described in section 3.4. To improve the performance, we use the socket pair to calculate the hash value to find the correct connection in the connection hash table. After packet preprocessing, the main thread updates connection information such as the sequence number, virus scan buffer and state. Then it calls a thread from the corresponding application-specific thread pool to handle the application protocol. The flow chart in section 3.6 describes the procedure in this application thread. Since main thread and the application threads both access the shared data space, a mutex lock mechanism must be implemented to avoid the race condition. In this work, three application modules are implemented: HTTP, FastTrack and MSN.

Chapter 5 Performance Evaluation

5.1 Benchmarking Environment

In this chapter, we perform various benchmarks on this system. Our system is installed on a PC with Pentium!!! 1G CPU, 512MB SDRAM and 20GB hard disk. Figure 6 illustrates the benchmark environment. In this environment, there are two HTTP clients and three Web servers. Each client creates one hundred threads and each thread downloads a 2MB files from theses three Web servers. This means that these two clients download totally 4GB data from the Web Servers through our system.

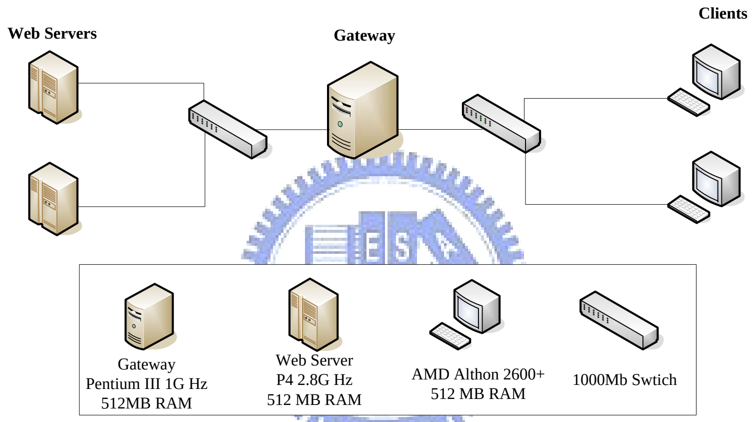


Figure 6: Benchmarking environment

There are two reasons why we use HTTP traffic instead of real P2P traffic to benchmark our system. First and the most important reason is that there are no such benchmark tools which can generate P2P traffic. The second is that many P2P applications like FastTrack and Gnutella use HTTP protocol to transfer files. Therefore, using HTTP traffic to simulate P2P traffic is acceptable.

5.2 Throughput and CPU Utilization

Throughput and CPU utilization are two important performance measures of a gateway system. Our system integrates many components. The following configurations are compared to understand the impact on performance from each component.

(1) NAT: System with pure NAT function.

(2) NAT+libipq: Besides NAT, every packet is queued in the kernel and duplicated to the proxy. The proxy just tells the kernel to pass the packets without any further processing.

(3) NAT+libipq+L7: Besides NAT+libipq, L7-filter is enabled with 20 rules. The entire process is similar to NAT+libipq. The only difference is that only HTTP are processed. This configuration is used to assess the performance impact of L7-filter.

(4) P2P Proxy + Filter: P2P Proxy is our system, which integrates NAT+libipq+L7 and implements the system described in Chapter 4. This configuration enables filtering transferred files according to the file name.

(5) P2P Proxy + Log File: P2P Proxy with the auditing function on transferred files. It records the transferred files into the file system.

(6) P2P Proxy + Virus Scan: P2P Proxy with the virus scanning function on transferred files.

(7) P2P Proxy + Filter + Log File + Virus Scan: P2P Proxy with all the above functions enabled.

Figure 7 and 8 show the throughput and CPU utilization, respectively, under all configurations. Figure 8 also plots the entire CPU usage but also shows the CPU usage of the kernel. In a gigabit network environment, pure NAT can reach the throughput about 280.04Mb/s. However, NAT+libipq reduce the throughput to 159.37Mb/s and the CPU has been fully used. This means copying packets from the kernel to the user space is a significant overhead. If the L7-filter is turned on, the throughput decreases obviously to 91.34Mbps and the CPU utilization most spent in the kernel. Enable filtering function does not influence the throughput too much. This is because the HTTP protocol is simple. But for those complex application protocols like MSN which need more processing, such as base64 [19] encoding and decoding, we believe that the influence on performance would be more obvious. The auditing functions does not influence the throughput much either. This result surprises us because what we believe is that the file access may cause large overhead. Therefore, we test the disk speed and the result shows that the speed of disk can reach 472Mb/s under sequential writing. This speed is much higher than the maximum throughput of our system, which explains why the influence is marginal. If virus scanning is enabled, the throughput decreases dramatically to 20.52Mbps and the proxy dominates about 70% of CPU utilization. Both L7-filter and virus scanning influence the throughput apparently and the key part of both is string matching. Hence, string matching is considered as the major bottleneck of this system.

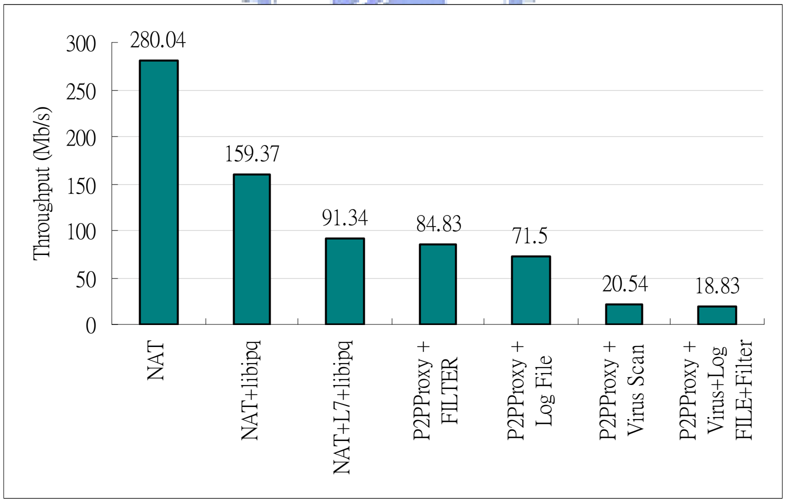


Figure 7: Maximum Throughput

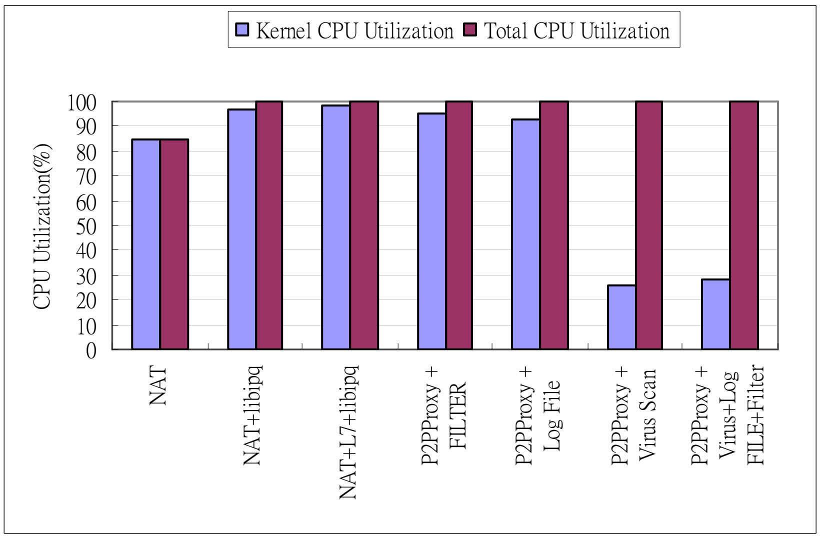


Figure 8: CPU Utilization

5.3 Internal Benchmarking

To further identify the bottlenecks of this system, we examine the execution time of each step in the entire packet processing flow with all functions turned on. Measuring execution time is performed by calculating the difference of time-stamps taken by the gettimeofday() system call before and after the code segments. Figure 9 (a) and (b) illustrate the basic packet processing steps with handling and without handling the head-of-line blocking problem. In the basic case where head-of-line might happen, M1 to M9 are processed sequentially. Handling the head-of-line blocking increases the throughput by a marginal 2Mb/s. Why the improving of throughput is marginal is because M7 dominates the throughput. In our observation, under our benchmarking environment, most connections are needed to perform virus scanning simultaneously. This means most packets are needed to wait the finish of virus scanning and only fewer packets can be processed during the period of virus scanning. Figure 9(c) presents the execution time in percentage. The result confirms that virus scanning is the key bottleneck since the time spent on virus scan is 200~800 times of other steps.

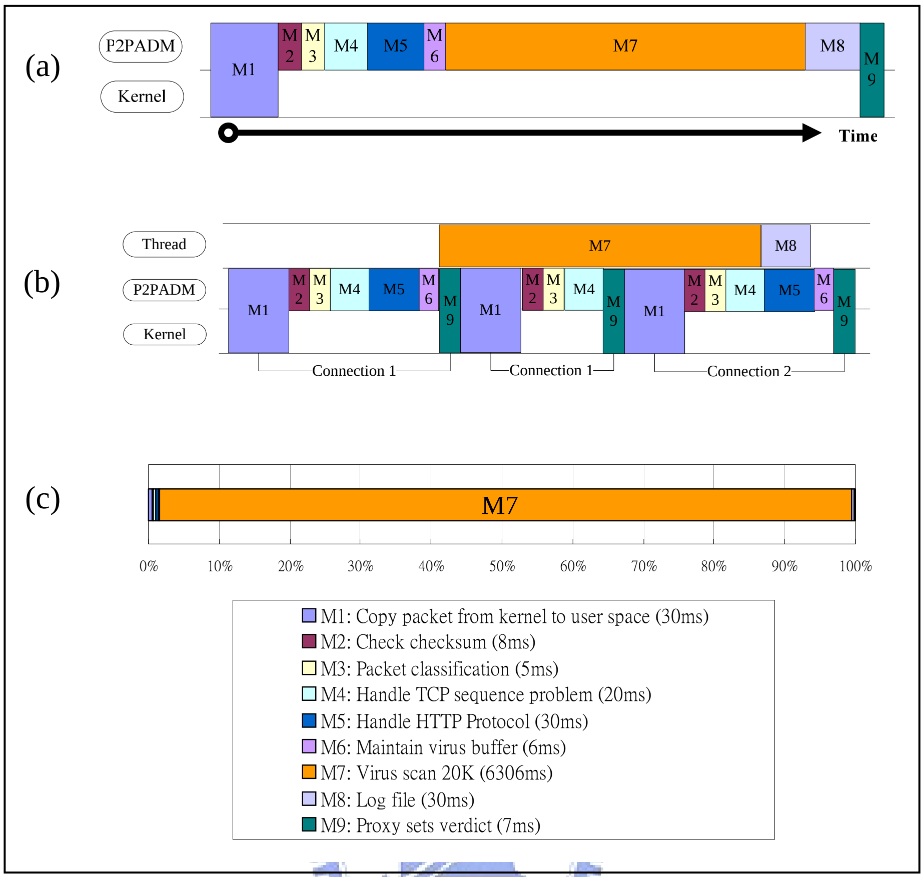


Figure 9: Internal benchmark result

5.4 Buffer Size

In this system, a buffer is allocated for each connection and virus scanning is only performed when the buffer is full. A small buffer takes less time to perform virus scanning, but it needs to be scanned frequently at the cost of calling more function calls. We evaluate the benchmark on throughput, in Figure 10, with buffer size form 1KB to 120KB to find out the best buffer size. Figure 10 shows the throughput with various buffer sizes. Choosing the small buffer size will involve too many virus scanning function calls. This will increase the system overhead and decrease the throughput. However, choosing a big buffer size will occupy too much memory and increase the packet queuing time when performing virus scanning. This may let the sending think that TCP congestion happens and slow down the transmission. Therefore, considering the memory size and the throughput, buffer size 20KB~40Kb is considered the best choice herein.

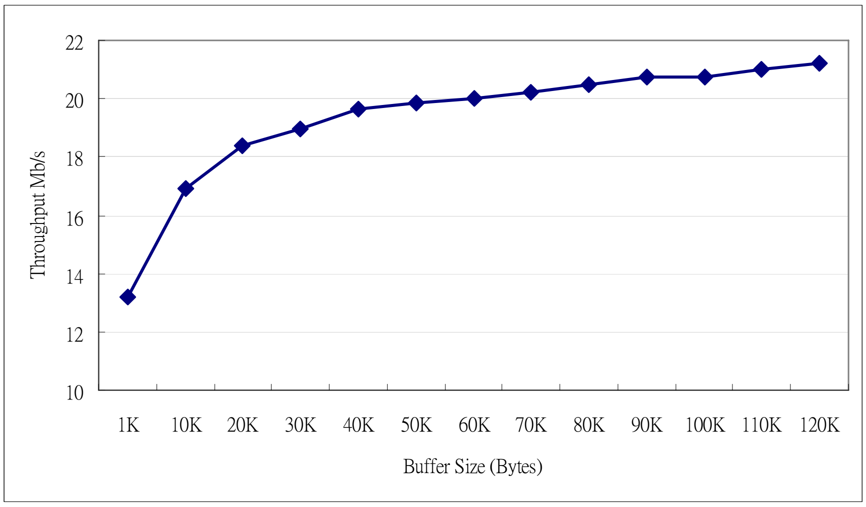


Figure 10: Throughput influence with different buffer size

5.5 Comparison with Conventional Proxy Architecture

This work also implements a simple HTTP proxy which uses the port redirection method to perform the same content management objectives like P2P Proxy. The only difference between P2P Proxy and this HTTP proxy is the traffic redirection mechanism. Theses two systems are compared, in Figure 11 and Figure 12, with functions turned on individually. Why the figure 12 only shows the kernel CPU utilization is because the CPU has been fully used on each situation. The port redirection method has higher throughput than P2P Proxy. Except virus scanning, the difference in throughput is about 40Mb/s, mostly due to connection classification. But if virus scanning is enabled, the difference is tiny. This means that the overhead of virus scanning is far larger than the connection classification. We suggest using port redirection method to manage fixed-port traffic and apply our mechanism to handle dynamic-port traffic.

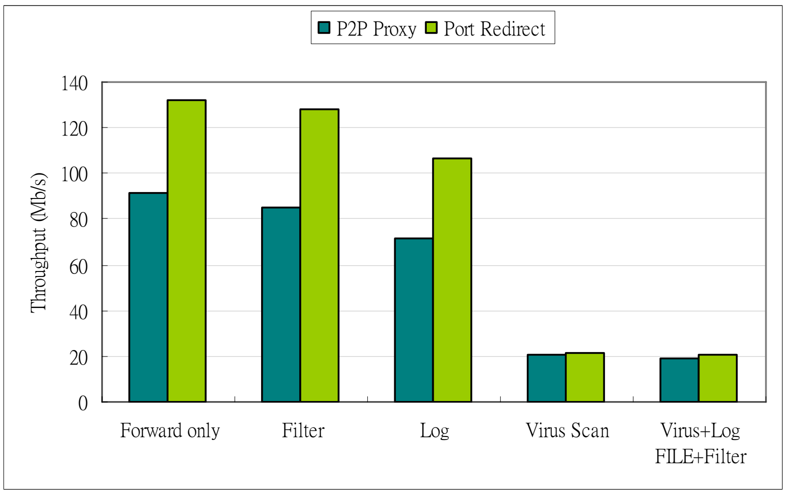


Figure 11: Maximum throughput of P2P Proxy and port redirect method

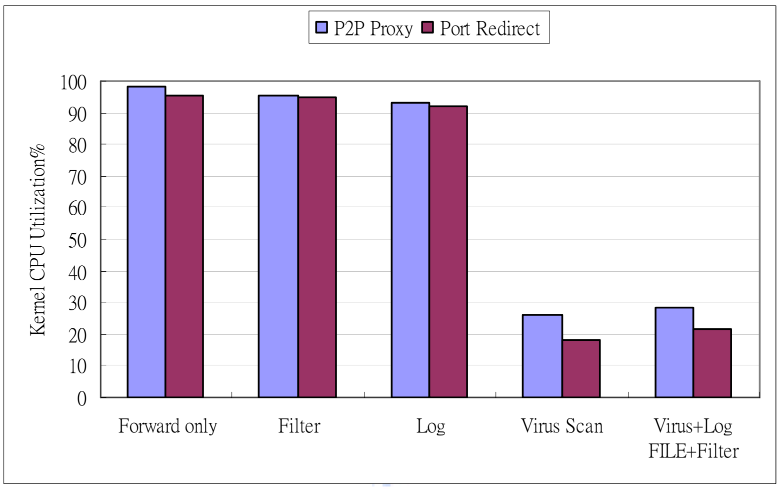


Figure 12: Kernel CPU utilization of P2P Proxy and port redirect method

Chapter 6 Conclusions and Future Research

This work presents a novel gateway architecture for managing peer-to-peer traffic with dynamic ports. The main challenge in this system is how to redirect the already established connection from the kernel to the user space to perform complex content filtering. We propose a *P2P Proxy Mechanism* which uses two packet queues in the kernel and a multiple-threaded user-space proxy called *P2P Proxy* to solve this problem. *P2P Proxy* uses modified *libipq* to communicate with the kernel and control packets between these two packet queues. *P2P Proxy* also solves the out-of-order problem of TCP sequence, the head of line blocking problem and the segmented virus signature problem.

The external benchmarking results indicate that virus-scanning influences the performance most. With and without enabling virus-scanning, the throughput can achieve 20.52MB/s and 84.83Mb/s, respectively. From the internal benchmarking, we confirm that string matching is the key bottleneck of this system since the time spent on string matching is 200~800 times of other steps. Comparing with port-redirect proxy, the connection classification and redirection impact the throughput about 40Mb/s and increase the CPU utilization too. For virus scanning, we also argue that considering memory size and throughput, the medium-size buffer size 20K-40K is suggested.

Since the string matching of content processing is the main bottleneck, to scale up the process, the string matching operations can be offloaded to an accelerator or ASIC. After comparing with port-redirect proxy, we suggest to use hybrid architecture which integrates port-redirect proxy and our system to handle both fix-port traffic and dynamic-port traffic. But how to let these two architectures to cooperate well is still a challenge. Besides these, as we know, there are many attack traffic exist on the

Internet. What kinds of attack will impact our system and how to avoid them is still a problem. Our future work will focus on these issues.

References

[1] S. Sen, O. Spatscheck, and D. Wang. Accurate, Scalable In-Network Identification of P2P Traffic Using Application Signatures. In *Proceedings International WWW Conference*, New York, USA, 2004.

[2] Thomas karagiannis, Andre Broido, Michalis Faloutsos and Kc claffy. Transport Layer Identification of P2P Traffic. In *ACM. SIGCOMM/USENIX Internet Measurement Conference (IMC 2004)*, Italy, October, 2004

[3] L7-filter. http://l7-filter.sourceforge.net/.

[4] Libipq. http://www.netfilter.org/documentation/FAQ/netfilter-faq-4.html

[5] A. Gerber, J. Houle, H. Nguyen, M. Roughan, and S. Sen. P2P The Gorilla in the Cable. In *National Cable & Telecommunications Association (NCTA) 2003 National Show*, Chicago, IL, June 2003

[6] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy. An Analysis of Internet Content Delivery Systems. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[7] S. Sen and J. Wang. Analyzing peer-to-peer traffic across large networks. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, Marseilles, France, November 2002

[8] IPP2P. http://rnvs.informatik.uni-leipzig.de/ipp2p/index_en.html.

[9] Netfilter. http://www.netfilter.org/.

[10] O. Spatscheck, J, Hansen, J, Hartman, and L, Peterson, "*Optimizing TCP forwarder performance,*", Transactions on Networking, 8(2):146--157. IEEE; ACM, April 2000.

[11] D. A. Maltz and P. Bhagwat, "*TCP Splicing for Application Layer Proxy Performance*," IBM Research Report RC 21139, March 1998.

[12] D. C. Schmidt, T. Harrison, and N. Pryce, "*Thread-Specific Storage -- An Object Behavioral Pattern for Accessing perThread State Efficiently*," in The 4 Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-34), September 1997.

[13] eDonkey. http://www.edonkey2000.com/.

[14] BitTorrent. http://bitconjurer.org/BitTorrent/.

[15] FastTrack. http://www.slyck.com/ft.php

[16] Guntella. http://www.gnutella.com/.

[17] MSN. http://www.msn.com/.

[18] ClamAV. http://www.clamav.net/

[19] Base64. http://www.ietf.org/rfc/rfc3548.txt?number=3548