國立交通大學

資訊科學與工程研究所

碩士論文

在點對點閘道器上整合與加速以內容為基礎的
辨別及管理系統

Integrating and Accelerating Content Classification
and Management at P2P Gateways

研 究 生：張朝江

指導教授：林盈達 教授

中華民國 九十五 年六月

在點對點閘道器上整合與加速以內容為基礎的辨別及管理系統

Integrating and Accelerating Content Classification and Management at P2P Gateways

研 究 生： 張朝江　　Student : Tsao-Jiang Chang

指導教授： 林盈達　　Advisor : Ying-Dar Lin

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2006

HsinChu, Taiwan, Republic of China

中華民國九十五年六月

在點對點閘道器上整合與加速以內容為基礎的辨別及管理系統

學生：張朝江　　　　指導教授：林盈達

國立交通大學資訊工程系

摘要

點對點軟體使用動態連接埠來隱藏自己,傳統利用連接埠來重導連線的代理伺服器架構已無法有效管理點對點連線。藉由檢查應用層的內容，一個在閘道器上的套件叫做 P2PADM 已經能夠管理點對點連線。此套件在核心空間進行連線辨別的動作,在使用者空間進行連線管理動作，因此內容置換和從核心空間到使用者空間的資料傳遞是必要的。我們提出一個叫做 kP2PADM 的新架構，此架構是將 P2PADM 套件從使用者空間搬移到核心空間的方式來改善效能，此工作最主要的困難是如何讓新架構能夠相容於 Linux 核心而不會造成核心恐慌。外部測試顯示 kP2PADM 的吞吐量比原始的 P2PADM 高出 88.85Mbps。此外，此工作亦點出 P2PADM 的兩個弱點:(1)重覆連線的問題，(2)當封包遺失時會發生多餘的延遲，針對此兩個弱點我們提出兩個解決方法:(1)內容快取，(2)快速通過。

關鍵字: 點對點, 核心模組, 內容快取, 快速通過

Integrating and Accelerating Content Classification and Management at P2P Gateways

Student: Tsao-Jiang Chang Advisor: Dr. Ying-Dar Lin

Department of Computer Science

Nation Chiao Tung University

Abstract

Peer-to-peer (P2P) software runs over dynamic ports in order to disguise their existence. Conventional port-redirecting proxy architecture cannot manage P2P traffic effectively. A gateway-based P2P administration (P2PADM) architecture had been proposed for managing P2P traffic by inspecting content at the application layer. P2PADM executes connection classification in the kernel space and connection management in the user space. Context switch and data passing from the kernel space to the user space is necessary. We propose a new architecture called *kP2PADM* for improving the performance of P2PADM by moving the P2PADM package from the user space to the kernel space. The main challenge in this work is how to move the code to the kernel space compatibly with Linux kernel without panicking the kernel. The external benchmarking reveals that the throughput of kP2PADM is 88.85 Mbps higher than that of P2PADM. This work also addresses two weaknesses of P2PADM: (1) *reconnection issue* and (2) *redundant delay due to packet loss*. Two solutions are proposed for the two weaknesses: (1) *connection cache* and (2) *fast pass*.

Keywords: peer-to-peer, kernel module, connection cache, fast pass

Acknowledgements

Many People have helped me with this thesis. I deeply appreciate my thesis advisor, Dr. Ying-Dar Lin, for his intensive advice and instruction. I would like to thank all the classmates in High Speed Networks Laboratory for their invaluable assistance and suggestions.

Finally, I thank my Father and Mother for their endless love and support.

Contents

List of Figures..vi

Chapter 1 Introduction..1

Chatper 2 Related Works..3

2.1 Introduction to P2PADM..3
2.2 Problems of P2PADM..4
2.3 Related in-kernel Solutions..4

Chapter 3 System Architecture Design..6

3.1 Solutions and the Proposed Architecture..6
3.2 In-kernel Management Architecture..8
3.2.1 Possible Approaches to Move User Modules to Kernel Space 8
3.2.2 Functional Modules in kP2PADM 9
3.2.3 Packet flow in kP2PADM.. 9
3.3 Connection Cache..10
3.4 Fast Pass..11

Chapter 4 Performance Evaluation..13

4.1 Benchmarking Environment..13
4.2 Comparison with Original Proxy Architecture..14
4.2.1 Throughput and CPU Utilization of kP2PADM 14
4.2.2 Throughput and CPU Utilization of kP2PADM plus the Connection Cache 17
4.2.3 Evaluation of Fast Pass 18
4.2.3.1 Benchmarking Environment 18
4.2.3.2 Impact from Out-of-order Packets on Performance 19
4.3 Internal Benchmarking 19

Chapter 5 Conclusions..20

References..21

List of Figures

FIGURE1: IMPLEMENTATION OF P2PADM SYSTEM ARCHITECTURE 3
FIGURE2: THE PROPOSED ARCHITECTURE OF KP2PADM 7
FIGURE3: PACKET FLOW IN THE NEW ARCHITECTURE 10
FIGURE4: BENCHMARK ENVIRONMENT OF KP2PADM..................................13
FIGURE5: THROUGHPUT OF P2PADM AND KP2PADM.....................................15
FIGURE6: CPU UTILIZATION OF P2PADM AND KP2PADM 15
FIGURE7: DIFFERENCE OF THE PACKET FLOW IN P2PADM AND KP2PADM 16
FIGURE8: THROUGHPUT OF KP2PADM PLUS CONNECTION CACHE 17
FIGURE9: CPU UTILIZATION OF KP2PADM PLUS CONNECTION CACHE....18
FIGURE10: BENCHMARKING ENVIRONMENT FOR FAST PASS.....................18
FIGURE11: TRANSFER TIME WITH FAST PASS AND WITHOUT FAST PASS IN DIFFERENT PACKET LOSS RATE 19
FIGURE12: INTERNAL BENCHMARK RESULT...20

Chapter 1 Introduction

Over the last few years, peer-to-peer (P2P) file sharing has grown astonishingly to dominate the Internet traffic [1]. Managing P2P traffic efficiently and effectively thus becomes an important issue. System administrators used to manage Internet traffic by classifying it according to fixed well-known port numbers. The management includes blocking traffic of specific applications or redirecting the connections to a proxy that performs various kinds of content filtering such as virus scanning. Nonetheless, the classification 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. Therefore, P2P applications should be classified according to the signatures in the application-layer messages [2]. The classification is traditionally executed in the kernel space because it is simple signature matching from the first few bytes of the content. However, the management such as filtering transferred files and scanning viruses on P2P shared files involves complex content processing of the data assembled from the packets. Thus it looks natural to be executed in the user space.

Although executed in the user space, the P2P management tools, such as InstantScan and P2PADM [3], need to exchange data between the kernel space and the user space. The data exchange, however, is a costly overhead involving the memory copy between the kernel space to the user space. In fact, the overhead also exists in web server packages, e.g. HTTPd. To reduce the overhead, an in-kernel package kHTTPd *(http://www.fenrus.demon.nl/)* moves HTTPd into the kernel space to directly handle requests in kernel. This approach avoids the data exchange and indeed provides higher performance than a user-space HTTP daemon.

This work attempts to avoid the data exchange in P2PADM. We move the P2PADM package from the user space to the kernel space. The implementation is

based on P2PADM because of the availability of its source code. We also address two weaknesses of P2PADM: the *reconnection* issue and *non-deterministic delay due to out-of-order packets*. The reconnection issue occurs because some P2P applications, say eDonkey, or users will persistently try to reconnect to the peers while P2PADM blocks their connection establishment. The reconnection keeps sending the same requests in a short period because P2PADM always blocks them. These useless requests will reduce the performance. Non-deterministic delay from out-of-order packets occurs because P2PADM must queue the out-of-order packets in order to handle those packets and so the packet delivery time to its peer is non-deterministic. For the reconnection issue, this work designs a connection cache to handle the packets for reconnection. For non-deterministic delay, the proposed architecture passes the out-of-order packets immediately.

The rest of this work is organized as follows. Chapter 2 introduces P2PADM and indicates its problems. Chapter 3 presents the proposed solutions and architecture. Chapter 4 discusses the performance of the proposed system. Chapter 5 concludes the study.

Chapter 2 Related Works

2.1 Introduction to P2PADM

P2PADM is a novel gateway architecture to manage P2P traffic. The management objectives in the architecture cover (1) connection classification 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 P2P traffic.

Fig. 1 illustrates the architecture of P2PADM. The kernel queues the packets of the classified connections identified by the L7-filter. A main thread in the proxy gets packets from the queue in the kernel by invoking the *libipq* library *(http://www.cs.princeton.edu/~nakao/libipq.htm)* and performs the pre-processing tasks, such as checksum examination, packet classification and TCP sequence handling. The main thread then calls a specific application thread to handle the tasks related to the application protocol. Each application thread is responsible for a specific connection and decides to pass or drop the packets in the connection.

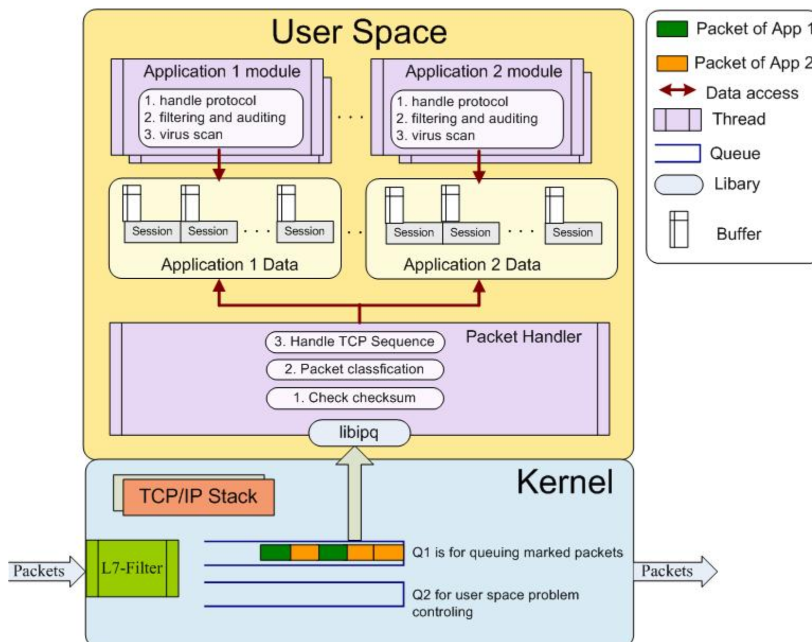


Figure 1. Implementation of P2PADM system architecture

2.2 Problems of P2PADM

According to the description in Section 2.1, P2PADM gets packets from the kernel queue by *libipq*. Libipq is a development library for iptables *(http://www.netfilter.org/projects/iptables/index.html)* and it provides an API to communicate with the *ip_queue* kernel module that registers with *Netfilter (http://www.netfilter.org/)* to pass packets between the kernel space and the user space. Therefore, P2PADM must do context switching between the kernel and user modes and copy data from the kernel space to the user space for managing P2P traffic. The impact of copying data can reduce the performance of P2PADM.

According to the previous benchmark result in [3], we are aware that the heavy use of *libipq* on P2PADM reduces the throughput by about 120 Mbps. Because *libipq* is responsible for copying data from the kernel space to the user space, reducing the heavy use of *libipq* can increase the throughput of P2PADM. We also find a few additional SYN packets sent by the P2P applications, say eDonkey, or users in the peer for reconnection while P2PADM blocks the establishment of a connection. They will be sent several times because the peer cannot establish the connection successfully. These useless SYN packets are always handled and dropped by P2PADM all the time, and reduce the performance of P2PADM.

Furthermore, we also find that there are some *non-deterministic delays from out-of-order packets*. All out-of-order packets cannot pass P2PADM because P2PADM blocks them for assembling them completely and manages them effectively. Because P2PADM is a transparent proxy to the peers, the non-deterministic delay not from the network transport but from the blocking of P2PADM should be avoided.

2.3 Related in-kernel Solutions

kHTTPd is an http-daemon for Linux and is different from other http-daemons in

that it runs within the Linux-kernel as a module. kHTTPd handles only static Web pages, and passes all requests for non-static information to a user-space Web server such as Apache. Since virtually all images are static and a large portion of HTML pages are also static, the improvement is significant. Static Web pages are not difficult to serve because the delivery of static objects from a Web server is simply a "copying file to network" operation. The Linux kernel is very good at this, and so as an in-kernel daemon, kHTTPd can gain better performance five times than other user-space http-daemons (http://www.fenrus.demon.nl/performance.html). Whether the complex management tasks on P2PADM can be performed entirely in the kernel in a similar way is interesting.

Chapter 3 System Architecture Design

3.1 Solutions and the Proposed Architecture

On P2PADM, the connections of P2P applications have been classified in the kernel space by L7-filter and managed in the user space. All the packets of each connection must pass through the user space, so the processing in the user space may become a bottleneck. Like *kHTTPd* that moves the code from a user-space daemon to a kernel-space module, this work also moves the code of P2PADM from the user space to the kernel space and then evaluates the improvement in performance.

Moreover, we design a *connection cache* to solve the *reconnection issue*. Because all packets of reconnection have the same (1) source IP address, (2) destination IP address, (3) destination port number (4) protocol id, and perhaps (5) source port number. The *connection cache* can easily identify a reconnection by keeping the five tuples of a blocked connection, and block it before P2PADM.

Furthermore, *non-deterministic delays from out-of-order packets* can be solved by duplicating the packets once in the gateway and fast passing the out-of-order packets to the destination instead of queuing them in P2PADM. The receiver can receive the out-of-order packets and send *triple ACKs* to the sender to invoke the retransmission. Because the retransmission is invoked by *triple ACKs* rather than *TCP timeout*, the non-deterministic delays will be shortened when the packets are lost.

Fig. 2 illustrates the operation of the proposed architecture. The entire architecture is called *kP2PADM*. The letter ‘k’ in the prefix of *kP2PADM* is used to differ it from the previous one because most functional modules of the proposed architecture are in the kernel space.

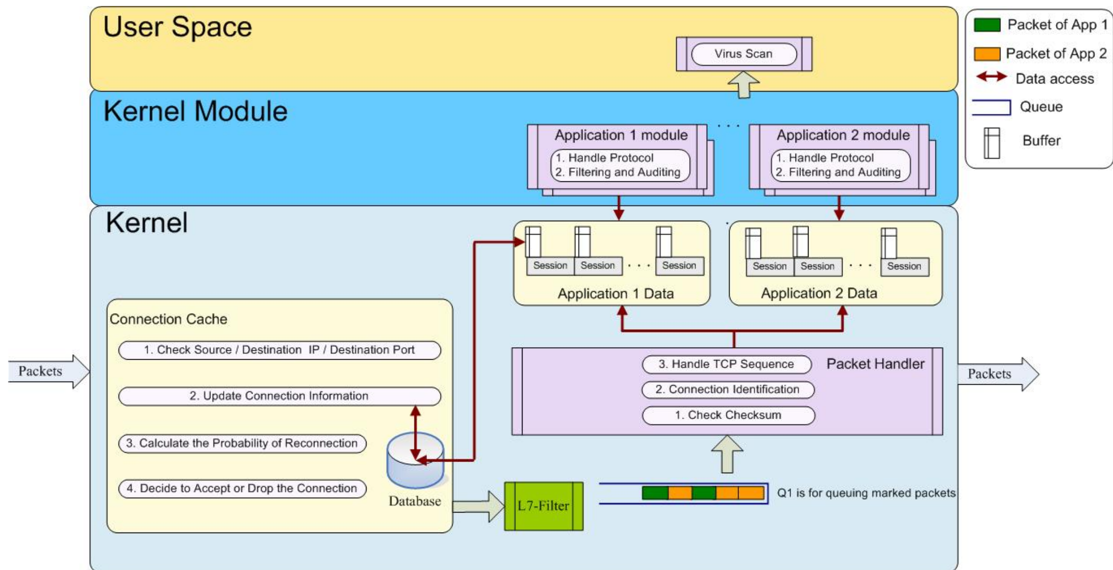


Figure 2. The proposed architecture of *kP2PADM*

In the beginning, all packets can pass through the in-kernel connection cache because the *connection cache* is empty. The L7-filter then performs connection classification in the kernel. The L7-filter collects at most the first eight packets to reassemble an application message and does signature matching. If the connection is identified by the L7-filter, it will be marked by a predefined application identifier. The kernel can filter the undesirable applications and do bandwidth control according to this predefined application identifier. The packets are then transferred to be pre-processed, say checksum check, connection identification, and TCP sequence handling. *kP2PADM* must occasionally call the *schedule* function to surrender the CPU control to other processes to avoid starvation. The *schedule* function is a Linux kernel function in *schedule.c* for process scheduling. The CPU control will come back to *kP2PADM* if no other processes demand the CPU. After *kP2PADM* finishes packet pre-processing and calls a specific AP module to handle the related packets. Each AP module is a kernel module responsible to set verdict to these related packets. All the handling of *kP2PADM* is in the kernel space except virus scanning. *kP2PADM* calls the *call_usermodehelper* function to invoke virus scanning in the user space, and blocks the Linux kernel until virus scanning is finished. To prevent from long

blocking, the file is scanned piece by piece. After scanning a piece of data, P2PADM calls the *schedule* function and may surrender the CPU control to the kernel or the other processes. The virus scanning of *kP2PADM* has been not implemented yet and to be completed in the future.

The *connection cache* is filled with a new 5-tuple value of a denied connection by *kP2PADM*. A reconnection of a denied connection will then be blocked by the *connection cache* rather than *kP2PADM* because the connection cache has recorded the denied connection.

3.2 In-kernel Management Architecture

3.2.1 Possible Approaches to Move User Modules to Kernel Space

There are two ways to move the code of P2PADM from the user space to the kernel space: one is coding functions in the *iptables* extended match module (*http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.html*) and the other is modifying the code of P2PADM to be a new kernel thread. The former codes the classification and management functions, such as connection classification, checksum check, TCP sequence handling, content filter, message log and the proposed connection cache in the *iptables* extended match module. A new filter rule including all the handling of *kP2PADM* is registered to the hook of *Netfilter* framework and all packets traverse through the filter rule. The latter creates a new kernel thread to run *kP2PADM*. The new kernel thread acquires packets from *ip_queue* that queues the packets identified by L7-filter and handles the operation of *kP2PADM* by itself. Therefore, the classification functions are coded in *iptables* extended match modules and the management functions are coded in the new kernel thread. We choose the latter in this work because the second way makes each function module simple and clear.

After choosing creating a new kernel thread to run *kP2PADM*, there are some implementation issues like how to transfer a system call from the user space to the kernel space? For example, the *read* system call is supported by the kernel for acquiring data from I/O device, but it cannot be called in *kP2PADM* because kP2PADM is a kernel process. A kernel process cannot call almost any system call because almost all system calls are implemented to be called by user programs and some handling in the system calls is unnecessary for kernel process, say data copy from the user space to the kernel space, and vice versa. Therefore, the *read* system call must be modified. Fortunately, the implementation of the *read* system call calls *vfs_read* for acquiring data from I/O devices and *vfs_read* is an *EXPORT_SYMBOL* kernel function [4]. An *EXPORT_SYMBOL* kernel function can be called by any kernel process. Therefore, *kP2PADM* can call *vfs_read* instead of system call *read*. Through this way of modifying user-space functions to kernel-space function, we can make P2PADM run in the kernel space.

3.2.2 Functional Modules in kP2PADM

The proposed architecture differs from P2PADM at the aspect of management. The packet pre-processing functions, such as checksum check, connection identification and TCP sequence handling, are moved to the kernel space, application protocol processing to a kernel module, and virus scanning left in the user space. The processing of application protocols is moved to a kernel module instead of the kernel because the application protocol processing may be sometimes updated and the modification of kernel module is faster than the modification of kernel. Leaving virus scanning in the user space is better than moving it to the kernel space because virus scanning takes much time and may block the kernel.

3.2.3 Packet Flow in kP2PADM

Fig. 3 illustrates the packet flow in *kP2PADM*. First, we create a new kernel

thread before the kernel invokes the *init* process. The kernel thread runs kP2PADM and is terminated when Linux is shut down. The in-kernel management architecture waits for new connections and calls the *schedule* function to surrender the CPU utilization to other processes to avoid starvation. After accepting a new connection, kP2PADM maintains the data structure of the connection socket. kP2PADM can do the I/O operations with the data structure rather than rely on the functions at the higher layers. After performing the pre-processing tasks, say packet classification and handling TCP sequence, kP2PADM signals the specific application thread to handle the packet. The application thread then sets verdict to the packet. The entire flow of an application thread is the same as that in P2PADM. The virus scanning of *kP2PADM* has been not implemented yet and to be completed in the future.

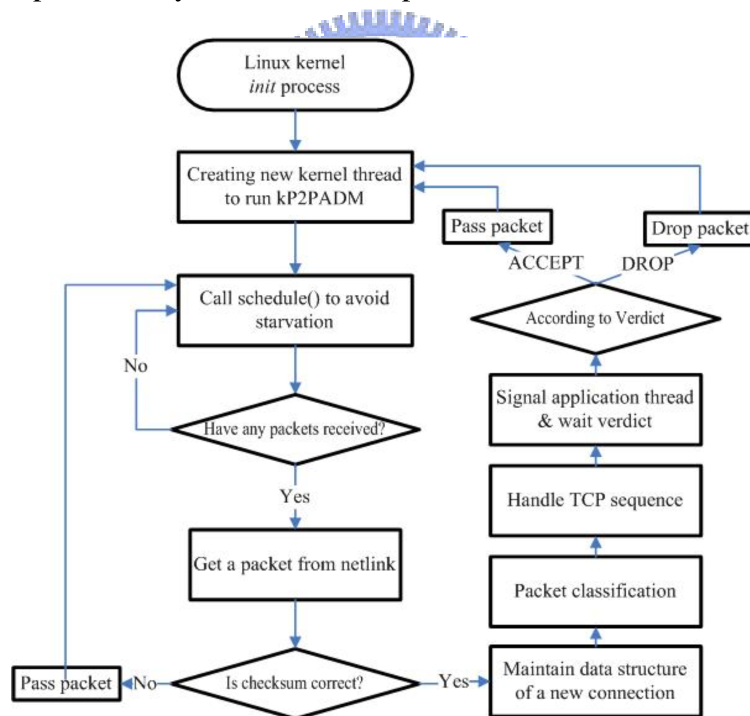


Figure 3: Packet flow in the new architecture

3.3 Connection Cache

To design the *connection cache*, we must consider that what information should to be kept. In the beginning, the packets in all connections can pass through the connection cache and be processed by *kP2PADM* because no connections have been

marked as denied ones. If *kP2PADM* decides to block a connection, then its *source IP address*, *source port number*, *destination IP address*, *destination port number*, and *protocol id* will be stored into connection cache. The packets having the same source IP address, destination IP address, destination port number and protocol id are viewed as in the reconnection, even though their source port numbers may be different. For example, BitTorrent (http://www.bittorrent.com/) changes to different source port number if connection is blocked. We believe that *kP2PADM* can identify reconnections through these tuples. A reconnection of a denied connection can be quickly dropped by the connection cache without being processed by *kP2PADM* and the performance of P2PADM can be improved.

3.4 Fast Pass

A more efficient way to handle the out-of-order packets is to duplicate them once in the gateway and pass them immediately so that the peer can receive the complete file early. If any packet is lost, the receiver can receive the out-of-order packets and send *triple ACKs* to the sender to invoke the retransmission rather than let these out-of-order packets be queued in gateway and the retransmission be invoked through TCP timeout. Because the retransmission is invoked by *triple ACKs* rather than *TCP timeout*, the non-deterministic delays will be shortened when the packets are lost. However, the retransmission may be redundant if the out-of-order packet is not made by packet loss. The redundant retransmission will decrease the throughput of kP2PADM. Besides, out-of-order packets passing without content filtering may escape the rule examination and result in false negatives. The probability of false negatives is very low in reality because kP2PADM still scans the out-of-order packet; nevertheless, it does not scan the content between packets. Fortunately, a signature between two packets is not frequent. Transfer time and false negatives are trade-off in

the design, and we will evaluate whether *fast pass* is a good design or not in this work

Chapter 4 Performance Evaluation

4.1 Benchmarking Environment

In this chapter, we perform various benchmarks on the kP2PADM system. *kP2PADM* is installed on a PC with Pentium III 1GHz CPU, 512 MB SDRAM and 20GB hard disk. Fig. 4 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 these three web servers. This means that these two clients download totally 1GB data from the Web servers through *kP2PADM*.

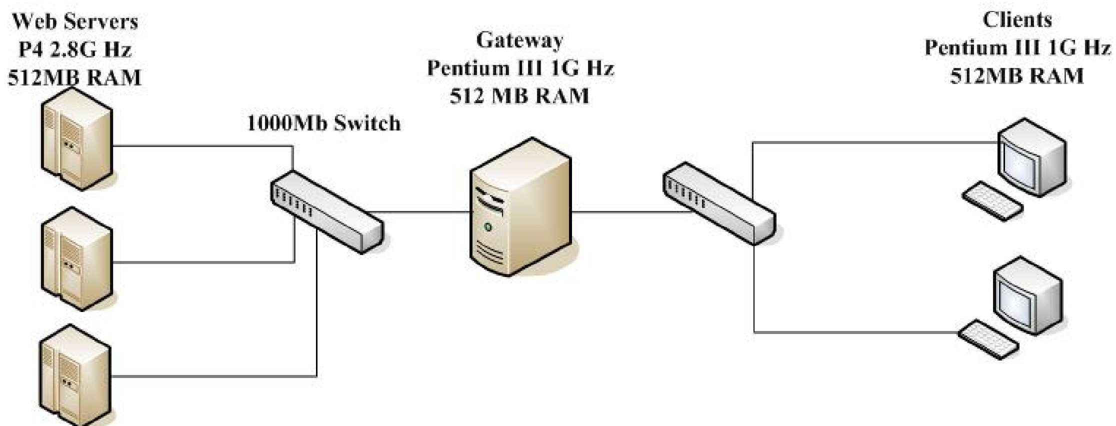


Figure 4: Benchmark environment of kP2PADM

There are two reasons why we use HTTP traffic instead of real P2P traffic to benchmark kP2PADM. First 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.

4.2 Comparison with Original Proxy Architecture

4.2.1 Throughput and CPU Utilization of kP2PADM

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

(1) *NAT*: the pure NAT function.

(2) *NAT + packet queue*: Besides NAT, every packet is queued in the kernel. kP2PADM just tells the kernel to pass the packets without any further processing.

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

(4) *P2P proxy + Filter*: P2P proxy integrates NAT + packet queue + L7 and all pre-processing of P2P management. 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.

Fig. 5 and 6 show the throughput and CPU utilization on P2PADM and kP2PADM under every configuration. Fig. 6 also plots not only the entire CPU utilization but also the CPU utilization for the kernel. In a gigabit network environment, pure *NAT* can reach the throughput about 266.13 Mbps on both P2PADM and kP2PADM. *NAT + packet queue* reduce the throughput to 155.24 Mbps and the CPU has been fully used by P2PADM, but on kP2PADM, *NAT + packet queue* only reduce the throughput slightly to 223.71 Mbps and the CPU utilization most spent in the kernel. The

throughput of kP2PADM is 68.47 Mbps higher than that of P2PADM.

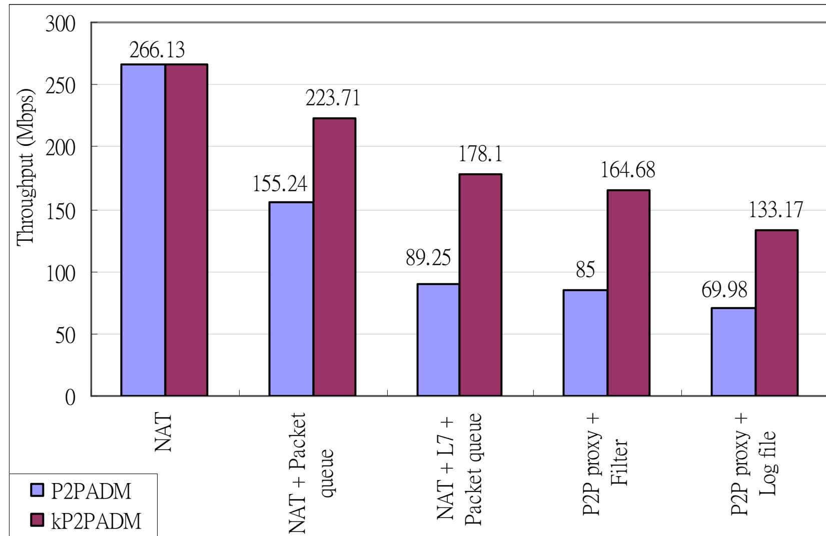


Figure 5. Throughput of P2PADM and kP2PADM

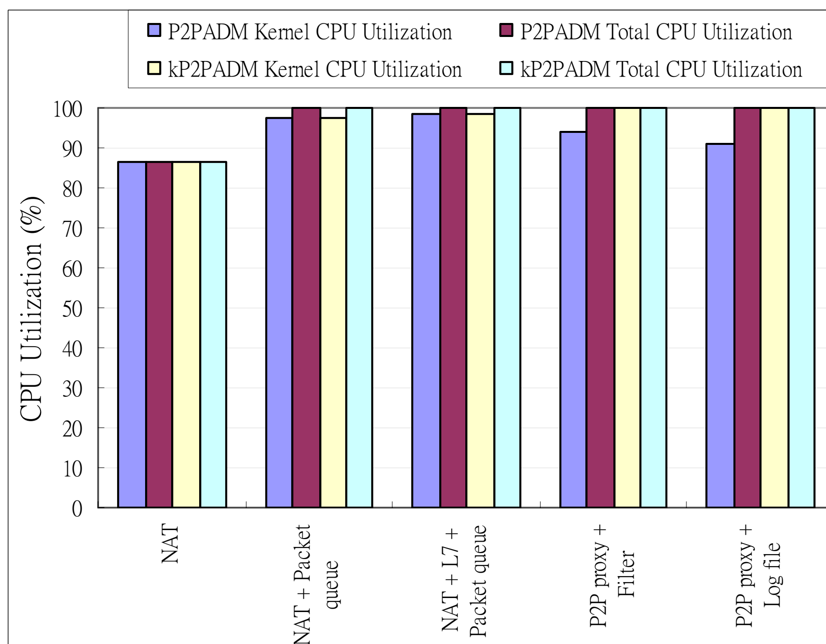
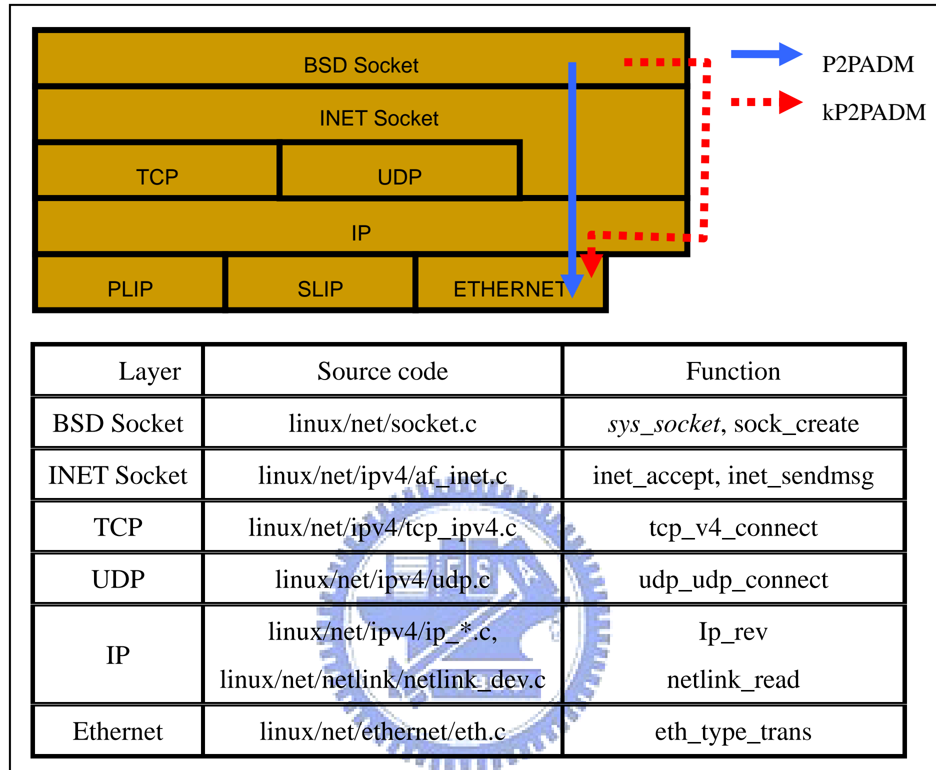


Figure 6. CPU utilization of P2PADM and kP2PADM

kP2PAADM is faster than P2PADM not only because coding in kernel space can reduce data copying from the kernel to the user space but also because it can reduce the number of calling functions. For example in Fig. 7, the solid line indicates the packet flow of P2PADM and the dashed line indicates the packet flow of kP2PADM.

While P2PADM creates a socket, it calls INET socket to acquire data, but kP2PADM calls netlink [5] directly instead of INET socket. Therefore, the number of functions kP2PADM called is fewer than P2PADM and kP2PADM can have higher performance than P2PADM.



Layer	Source code	Function
BSD Socket	linux/net/socket.c	*sys_socket*, sock_create
INET Socket	linux/net/ipv4/af_inet.c	inet_accept, inet_sendmsg
TCP	linux/net/ipv4/tcp_ipv4.c	tcp_v4_connect
UDP	linux/net/ipv4/udp.c	udp_udp_connect
IP	linux/net/ipv4/ip_*.c, linux/net/netlink/netlink_dev.c	Ip_rev netlink_read
Ethernet	linux/net/ethernet/eth.c	eth_type_trans

Figure 7: Difference of the packet flow in P2PADM and kP2PADM

If the *L7-filter* is turned on, the throughput decreases obviously to 89.25 Mbps on P2PADM and to 178.1 Mbps on kP2PADM. 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 encoding and decoding [6], we believe that the influence on performance would be more obvious. The *auditing functions* does not influence the throughput much either. On P2PADM, the throughput of *P2P proxy + log file* is 69.96 Mbps and 133.17 Mbps on kP2PADM. kP2PADM always dominates about 100% of CPU utilization while P2P proxy is on. It is because kP2PADM is implemented in the

kernel space and the kP2PADM always occupies the CPU for benchmark. It means that surrender the CPU timely to other processes for kP2PADM is very important, otherwise the kernel will be blocked by kP2PADM for a long time.

4.2.2 Throughput and CPU Utilization of kP2PADM plus the Connection Cache

Fig. 8 and 9 show the throughput and CPU utilization on kP2PADM as connection cache is enabled. In the experiment, we set a policy on kP2PADM to block all packets from one of two clients. This policy forces the blocked client to keep sending reconnections. The connection cache can increase the throughput by about 15%. The CPU utilization always reaches about 100% because all handlings of kP2PADM are coded in the kernel space except for virus scanning, so the CPU utilization will be occupied by kP2PADM.

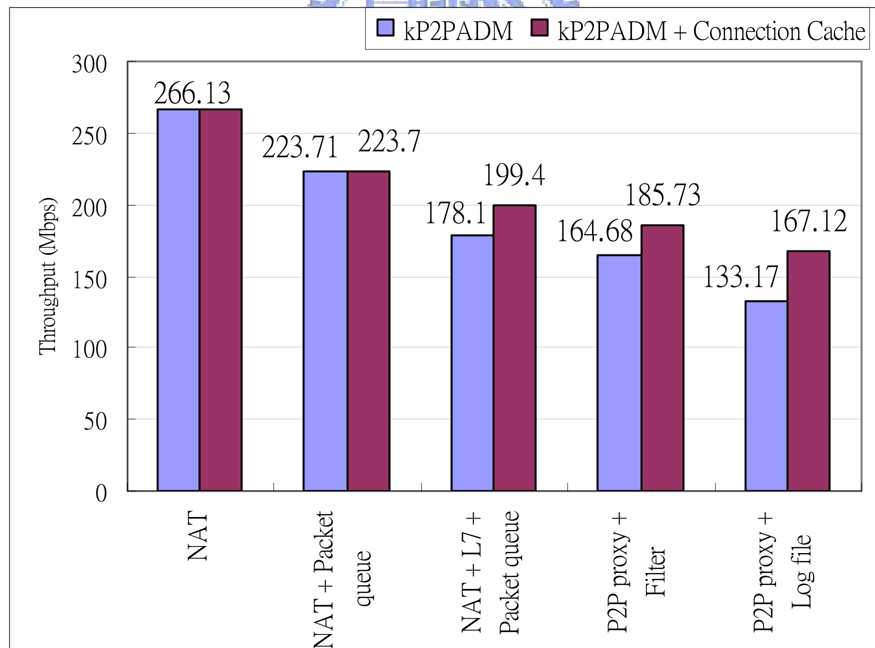


Figure 8. Throughput of kP2PADM plus connection cache

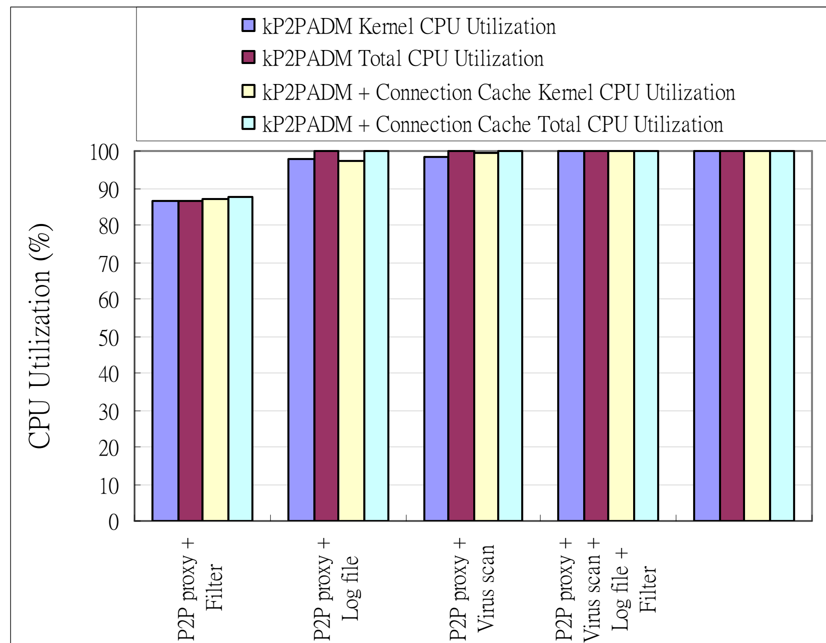


Figure 9. CPU utilization of kP2PADM plus connection cache

4.2.3 Evaluation of Fast Pass

4.2.3.1 Benchmarking Environment

To emulate packet loss and out-of-order packets, we install a WAN emulator called NIST Net from National Institute of Standards and Technology (NIST) [7] on Linux. NIST Net allows a single Linux PC to act as a router to emulate a wide variety of network conditions, say packet loss, out-of-order packets, transmission delay, and so on. Fig. 10 illustrates the benchmark environment. A NIST Net stands before kP2PADM and emulates packet loss and delay. A notebook serves as the FTP client to request about 300 MB files from the FTP server.

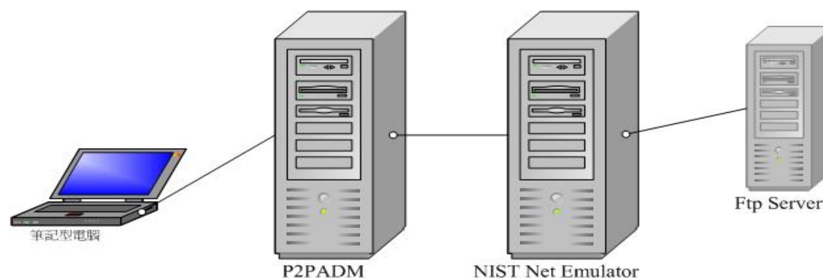


Figure 10. Benchmarking environment for fast pass

4.2.3.2 Impact from Out-of-order Packets on Performance

Fig. 11 shows the transfer time with *fast pass* and without *fast pass* in different packet loss rate, respectively. The packet loss rate is from 0% to 5% to simulate the real environment [8]. Fast pass can reduce the transfer time between ftp client and ftp server. In the benchmarking result, we have two observations: (1) the higher the packet loss rate is, the more the transfer time between with fast pass and without fast pass can shorten and (2) the longer the delay is, the more the transfer time can be reduced. The first observation is because the times of queuing in the gateway is more high if more packet loss rate, so transfer time will more larger. The second observation is because the queue time in the gateway is longer if the delay of each packet is longer. Shortly speaking, fast pass can reduce the more transfer time while the delay time and drop rate is larger.

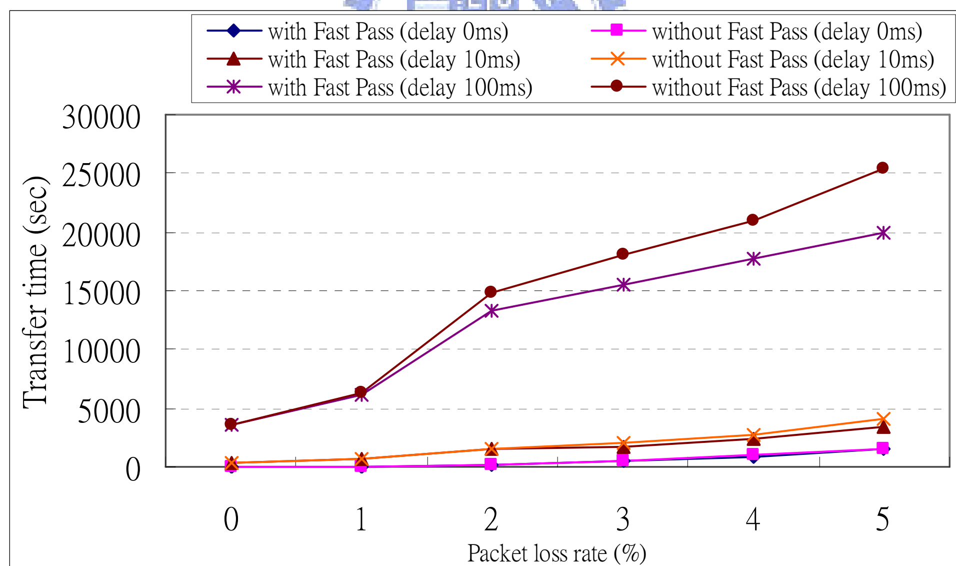


Figure 11: Transfer time with Fast Pass and without Fast Pass in different packet loss rate

4.3 Internal Benchmarking

To further identify the improvements and the bottlenecks of kP2PADM, 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 do_gettimeofday() kernel function before and after the code segments. Figure 12 (a) and (b) illustrate the internal benchmark of P2PADM and kP2PADM. Moving code from the user space to the kernel space can reduce the time of all steps. The most time-reduction is the time of getting packets and it shows that kP2PADM takes 5 ms to get packets. The least time-reduction is the time of handling HTTP protocol. Why the improving of handling HTTP protocol is marginal is because it takes much time to processing HTTP protocol rather than data passing between the kernel and the user space.

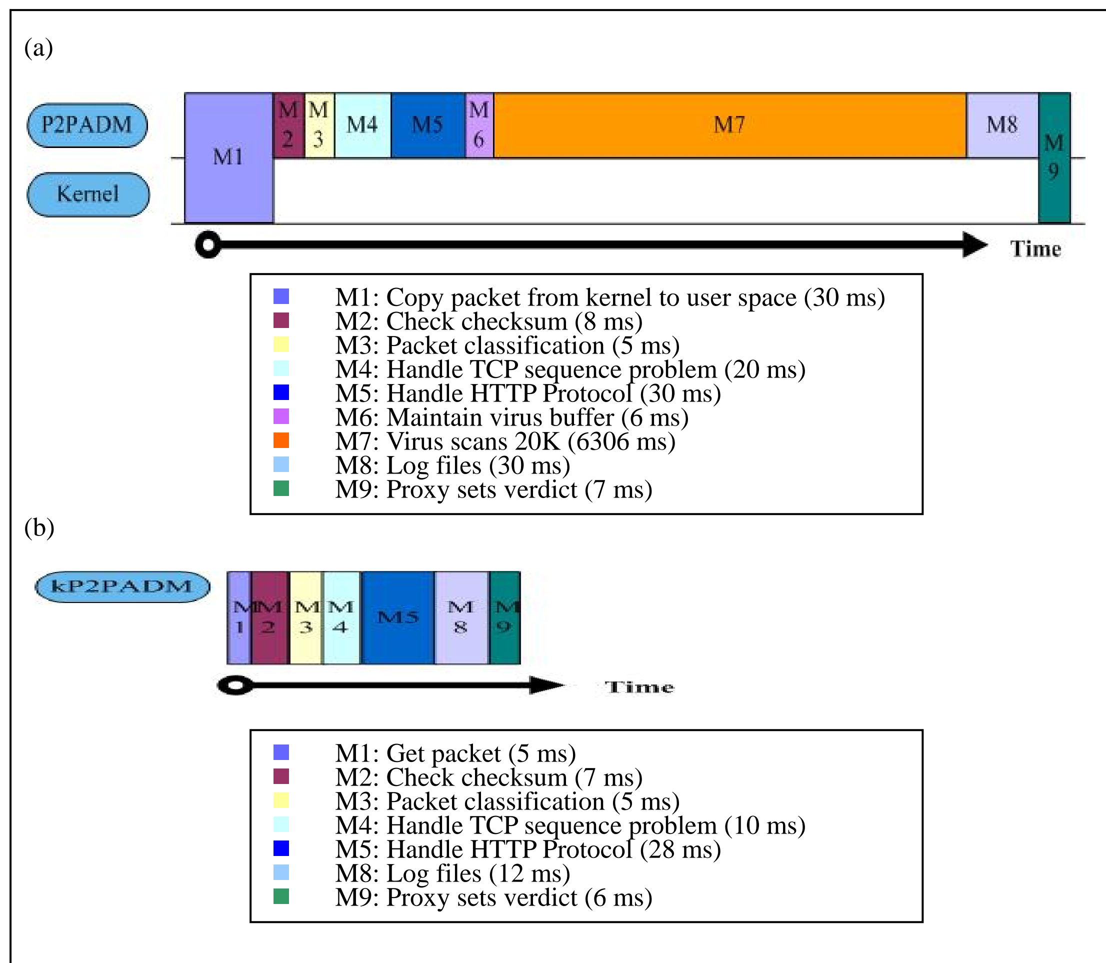


Figure 12: Internal benchmark result

Chapter 5 Conclusions

This work presents the improvement of the P2PADM performance by moving the code of management from the user space to the kernel space. The main challenge in this work is how to move the code to the kernel space compatibly with Linux kernel without panicking the kernel. The new architecture, kP2PADM, must maintain some kernel-space data structures by itself, say structure *sock* to get data form packets effectively rather than get data through socket description in P2PADM. Besides, kP2PADM is responsible for scheduling the kernel because it is a kernel process. A kernel process occupies the CPU unless it surrenders the CPU voluntarily. kP2PADM also solves two weaknesses of P2PADM: the *reconnection issue* and *non-deterministic delays from out-of-order packets*. Through *connection cache* and *fast pass*, we can increase the throughput of P2PADM and the transmission time.

The external benchmarking results indicate that in-kernel management improves the performance of P2PADM. With and without kP2PADM, the throughput can achieve 164.68 Mbps and 85 Mbps, respectively. The throughput of kP2PADM is 79.68 Mbps higher than that of P2PADM. The connection cache can increase the throughput by about 20 Mbps. The CPU utilization of kP2PADM always reaches to 100% that is because all the handling of kP2PADM is implemented in the kernel space except for virus scanning. Therefore CPU always blocked by kP2PADM when benchmarking. Fast pass can reduce the more transfer time while the delay time and drop rate is larger.

References

[1] S. Sen, and Jia Wang, "Analyzing Peer-to-Peer Traffic across Large Networks," *IEEE/ACM Transactions on Networking,* vol. 12, no. 2, pp. 219-232, April 2004.

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

[3] Meng-Fu Tsai, "A Novel Gateway Architecture for Managing Dynamic Port Peer-to-Peer Traffic," Master thesis, National Chiao Tung Unviersity, Hsinchu, Taiwan, 2005.

[4] Daniel P. Bovet and Marco Cesati, "Understanding the Linux Kernel," 2nd Ed, pp. 695, O'REILLY, 2003.

[5] J. Salim, H. Khosravi, A. Kleen, A. Kuznetsov, "Linux Netlink as an IP Services Protocol," RFC 3549, July 2003.
[Online] available: http://www.ietf.org/rfc/rfc3548.txt?number=3549

[6] S. Josefsson, "The Base16, Base32, and Base64 Data Encodings," RFC 3548, July 2003.
[Online] available: http://www.ietf.org/rfc/rfc3548.txt?number=3548

[7] Mark Carson, Darrin Santay, " NIST Net – A Linux-based Network Emulation Tool," Computer Communication Review, ACM SIGCOMM, 2003.

[8] Maya Uajnik, Sue Moon, Jim Kurose and Don Towsley, "Measurement and Modeling of the Temporal Dependence in Packet Loss," Tech. Rep. UMASS CMPSCI 98-78, University of Massachusetts, Amherst, MA, 1998.