# kP2PADM: An In-kernel Gateway Architecture for Managing P2P Traffic

Ying-Dar Lin[1], Po-Ching Lin[1], Meng-Fu Tsai[1], Tsao-Jiang Chang[1], and Yuan-Cheng Lai[2]

[1]National Chiao Tung University
Dept. of Computer Science
Hsinchu, Taiwan 300
{ydlin, pclin, mftsai, tjchang}@cis.nctu.edu.tw

[2]National Taiwan Univ. of Science and Technology
Dept. of Information Management
Taipei, Taiwan 106
laiyc@cs.ntust.edu.tw

## Abstract

*This work presents an in-kernel gateway architecture on Linux, namely* kP2PADM*, for managing P2P traffic on dynamic ports. This design can effectively eliminate redundant data passing between the kernel space and the user space. The management functions include (1) classifying and filtering P2P traffic, (2) scanning viruses on shared files, (3) auditing chatting messages and transferred files, and (4) bandwidth control. Practical implementation issues and techniques in the system design are discussed herein. This design proposes a dual-queue architecture to handle packet reassembly and resolve head-of-line blocking. A connection cache accelerates handling the reconnection requests from the peers. The throughput can achieve up to 185.73 Mbps even with content filtering, and remains around 79.09 Mbps when virus scanning is enabled. The impacts of each management function and out-of-order packets on performance are also analyzed through the internal benchmarks.*

## 1 Introduction

Over the past few years, peer-to-peer (P2P) applications have grown astonishingly to dominate the Internet traffic [5, 7]. Like other types of Internet traffic, P2P traffic should be also properly managed. For example, a company would not allow the workers to share files via P2P applications in working hours. Therefore, central management of P2P traffic on the gateway is a promising solution. The management typically includes the following functions: (1) classifying and filtering P2P traffic, (2) scanning viruses on shared files, (3) auditing chatting messages and transferred files, and (4) bandwidth control. Conventional approaches that classify Internet traffic according to fixed port numbers, such as those on firewalls, no longer work because most P2P applications tend to hide themselves by encrypting the messages or running on dynamic ports [3]. P2P traffic can be detected by either examining packet payloads [6, 8] or analyzing the connection pattern at the transport layer [4], but managing P2P traffic on a transparent gateway is more complicated than detection. Efficiently handling the packet flow is essential to handle a large number of P2P connections on a gateway.

This work designs an in-kernel architecture, namely *kP2PADM*, to transparently manage P2P traffic on a gateway. This design is implemented in the kernel space of Linux to avoid the overheads of passing data between the kernel space and the user space. The L7-filter (`http://l7-filter.sourceforge.net`) acts as a connection classifier that identifies P2P signatures in the application-layer messages. After the classification, the packets in a P2P connection are redirected to a kernel module that performs functions such as packet reassembly and content filtering. Because raw packets may be out-of-order, lost or duplicated, the module implements the TCP reassembly function to handle these situations. Such time-consuming processing may cause *head-of-line* blocking in the kernel queue, in which the packets in the other connections are blocked behind those under examination. This work proposes a dual-queue mechanism to handle this situation. A modified queue handler, *ip_queue*, manages the packets in the dual queues of the kernel. The kernel module is multi-threaded. A main thread handles packet arrival, and the others handle specific application protocols and perform the desired content filtering.

This work also addresses two factors that could reduce the performance: useless reconnection requests from P2P applications and out-of-order packets. The reconnection requests may occur because some users or P2P applications

1-4244-0910-1/07/$20.00 

themselves will persistently attempt to reconnect to their peers in a short period of time when the gateway blocks their connection establishment. Handling these requests in the same classification process as that in their first attempt is wasteful. They should be blocked again soon. Besides, out-of-order packets also result in redundant packet retransmission because the gateway must queue these packets to maintain the order. The sender may consider this case as packet loss if its TCP retransmission timer expires or it receives three duplicated TCP ACKs due to the queueing, and thus it retransmits the packets unnecessarily. For the former issue, this architecture designs a connection cache to handle the packets for reconnection. For the latter, this design duplicates a copy of each packet in the kernel for ordering and reassembly, and then passes out-of-order packets immediately. We call this strategy *fast pass*.

This work discusses the practical issues of designing an efficient P2P gateway, and proposes several strategies to resolve the problems. The rest of this work is organized as follows. Section 2 overviews typical P2P applications and surveys related packages in this work. Section 3 presents the key ideas of the design and the system architecture. The implementation will be also detailed in this section. Section 4 presents the performance evaluation of this system and analyzes the result. The study is concluded in Section 5.

## 2 Survey of Related Works

### 2.1 Overview of P2P Applications

Table 1 summarizes the characteristics of popular P2P applications: BitTorrent (`http://www.bittorrent.com`), eMule (`http://www.emule-project.net`), FastTrack (`http://www.slyck.com/ft.php`) and Gnutella (`http://www.gnutella.com`). Besides, chatting and file transfer in instant messengers (IM), such as MSN (`http://messenger.msn.com`) and Skype (`http://www.skype.com`), also work in the P2P mode. Most P2P applications can run on dynamic ports to circumvent the filtering of firewalls. The port numbers have default values chosen by the applications, but they can be either configured by the users or changed by the applications later. Some applications, say Skype, can choose the ports of HTTP and HTTPS in case of connection failure [1], and thus it is impossible to tell which application is being used from only the port number. Therefore, deep packet inspection to identify the P2P applications is necessary.

These P2P applications may transfer files in two approaches: one is sequential transfer, in which a peer receives a file sequentially from another peer, and the other is segmented transfer, in which the segments of a file can be received in an out-of-order manner. If a file is either encrypted or transferred in out-of-order segments, performing virus scanning and auditing becomes very difficult. For example, a user may download a file with a laptop in two different locations. The downloaded content is composed of random fragments to the gateway in either location, and thus the content can only be reassembled by the laptop itself. We do not implement both functions for such applications that encrypt data or transfer data in an out-of-order manner. If the file name is visible, filtering can refer to the file name that contains specific keywords. Enterprises may not want employees to leak out confidential information via a chatting system such as instant messengers. Filtering sensitive keywords or recording the message is also needed. The *kP2PADM* architecture is designed to implement feasible management functions summarized in Table 2.

| applications | ST | EN | DP | FV | DFP |
|---|---|---|---|---|---|
| FastTrack | Yes | No | Yes | Maybe | 1214 |
| eMule | No | No | Yes | No | 4661-4665 |
| BitTorrent | No | No | Yes | Yes | 6881-6889 |
| Gnutella | Yes | No | No | Yes | 6346,6347 |
| MSN | N/A | No | Yes | Yes | 1863 |
| MSNFTP† | Yes | No | Yes | No | No default |
| Skype | Yes | Yes | Yes | No | No default |

**Table 1. The characteristics of P2P and IM applications.**
**(†MSNFTP is a file transfer protocol of MSN.)**
**ST=sequential transfer, EN=with encryption, DE=data encryption, DP=dynamic port, FV=filename visibility, DFP=default ports.**

| applications | CF | FT | VS | AU | BC |
|---|---|---|---|---|---|
| FastTrack | Yes | No | No | Yes | Maybe |
| eMule | No | No | No | Yes | No |
| BitTorrent | No | No | No | Yes | Yes |
| Gnutella | Yes | No | No | No | Yes |
| MSN | N/A | No | No | Yes | Yes |
| MSNFTP | Yes | No | No | Yes | No |
| Skype | Yes | Yes | Yes | Yes | No |

**Table 2. Feasibility of management functions for each P2P protocol.**
**CF=classification, FT=filtering, VS=virus scanning, AU=auditing, BC=bandwidth control**

### 2.2 Related Packages in the Architecture

Although some commercial P2P detection tools exists, say *p2pwatchdog* (`http://www.p2pwatchdog.com`), open-source packages are preferred to be integrated into

this work. Several packages, such as L7-filter and IPP2P (`http://www.ipp2p.org`), can be used to identify P2P traffic. They are both classifiers that inspect the packet payload in the Linux Netfilter subsystem (`http://www.netfilter.org`). The L7-filter uses Netfilter's connection-tracking module and checks only the content in the first eight packets of the application data after a connection is established. If the application data matches one of the signatures, it marks the entire connection as identified by the module. IPP2P checks every packet because it does not have a connection-tracking module. The signatures of IPP2P are hard-coded in its code, while the L7-filter loads signatures from the files. Because the L7-filter inspects fewer packets and dynamically loads signatures, it has higher performance and better flexibility than IPP2P. This work therefore chooses to integrate the L7-filter as the classifier in the proposed architecture.

# 3 Design of System Architecture

Because P2P connection classification involves examining application messages, a TCP connection between two peers must be established first. *After* the connection has been established, redirecting the connection from the kernel to the kernel module to perform content filtering is performed in the following steps. (1) The L7-filter performs connection classification and marking. (2)The packets are queued in the kernel and wait the verdict from the kernel module. (3) The kernel module handles packet reassembly and decides the verdict by content inspection according to the management objectives. (4) The head-of-line blocking and segmentation of virus signatures are handled.

## 3.1 Main Ideas of the Design

### 3.1.1 Connection Classification and Marking

The L7-filter collects at most the first eight packets to reassemble application messages and does signature matching. If the connection is identified as P2P traffic, it will be marked by a predefined protocol number that indicates the type of P2P traffic. The kernel can then filter undesirable traffic and control the available bandwidth of P2P traffic according to this number. Before the L7-filter finishes signature matching, the packets that contain important information such as the file name or size might have been already passed to the peer, but the kernel module may still need such information to take certain actions. To solve this problem, a special packet is created inside the kernel and the application data collected by the L7-filter are packed into this packet after successful matching. This packet is internally passed only to the kernel module for extracting the application data and further processing.

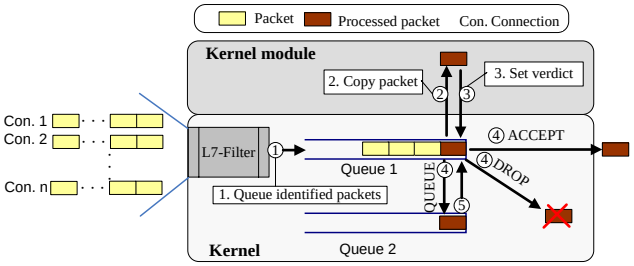


**Figure 1. Packet queueing and redirecting mechanism.**

### 3.1.2 Packet Queueing and Redirecting

Two packet queues Q1 and Q2 are created in the kernel to manage P2P traffic. All packets identified by the L7-filter are queued in Q1, while those unidentified packets are just passed through the gateway. The queued packets are passed to the kernel module and wait for the verdict from the module, which processes these packets in Q1 and sets the verdict for them sequentially. The verdict can be `ACCEPT`, `DROP` or `QUEUE`. `ACCEPT` means passing a packet to the peer, while `DROP` means dropping a packet. If the packet cannot 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 to wait temporarily. How the kernel module makes the verdict will be described in later subsections. Figure 1 illustrates this mechanism.

### 3.1.3 Packet Preprocessing

When the kernel module gets a packet from Q1, three tasks must be done before handling the P2P protocol. (1) The packet checksum is examined. If the checksum is incorrect, the module will not process the packet, but tell the kernel to pass it rather than drop it. The connection reliability is left to the two peers. (2) Packet classification identifies which connection the packet belongs to. (3) The correct sequence of the packet is handled before reassembly.

Because the kernel uses only a single queue to queue the packets of all marked connections, packets should be further classified based on the five tuples, i.e., source IP address, source port, destination IP address, destination port and protocol identifier. The packets may be out-of-order because the redirected packets do not pass through the TCP stack that can reorder the packets in the correct sequence. The kernel module therefore calculates the correct next sequence number, and checks the sequence number of the handled packet. If the sequence number of the packet is less than the correct one, this packet is a duplicated one and should be passed without further processing. If the sequence number of the packet is larger than the correct one,

this packet should wait until the appearances of all the packets with the sequence numbers less than this one. The kernel is instructed to move this packet temporarily from Q1 to Q2. If the sequence number of the packet is correct, this packet is processed and the out-of-order packets in Q2, if any, will be moved back to Q1. These packets are then reassembled by the kernel module.

### 3.1.4 P2P Protocol Processing

The P2P protocol is recognized according to the protocol number in the kernel, and handled after packet reassembly. The management functions are performed in this stage. A connection may be in one of the three states when the data are being transferred: (1) *initial state*: waiting for the request and response of data transfer, (2) *receiving state*: receiving the transferred data, and (3) *processing state*: performing virus scanning or content filtering on the received data.

The packets are checked based on the corresponding application protocol to examine whether the chatting message or the request of file transfer contains specific keywords. If a specific keyword is found, the kernel module instructs the kernel to drop the packet as well as the subsequent packets in the same connection, and then sends an RST packet to the source peer to tear down the connection; otherwise the kernel is instructed to pass this packet and record the chatting messages or file name if auditing is enabled. If the packet should not be blocked, the connection is marked to be in the receiving state. The following transferred data segments are reassembled as a file and virus scanning is performed until the file transfer is completed.

### 3.1.5 Virus Scanning for Shared File

A buffer is allocated for each connection. When the kernel module receives a packet, it first checks whether the buffer is full and whether the packet is the last one in the transferred file. If neither happens, the data segment is kept in the buffer and the module instructs the kernel to pass this packet; otherwise, the proxy performs virus scanning on the buffer. If a virus is found, the proxy tells the kernel to drop these packets, and send an RST packet to the peer to tear down the connection. Dropping these packets can destroy the file. If no virus found, the buffer is cleaned and the packets are passed. This method may still suffer two problems: the *head-of-line blocking* and the segmentation of virus signatures.

The head-of-line blocking occurs due to the time-consuming virus scanning. The subsequent packets queued in Q1 cannot be handled until virus scanning on the buffer is finished. This will constrain the throughput of the entire system. To solve this problem, when virus scanning is needed, the connection is marked to be in the processing

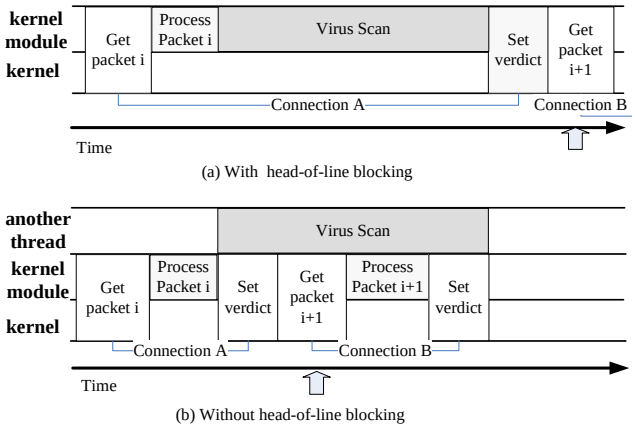


**Figure 2. Comparison of the packet processing with and without handling the head-of-line blocking.**

state, and another thread is called to perform virus scanning. The proxy tells the kernel to move subsequent packets of the same connection from Q1 to Q2. Therefore, the packets in other connections can be immediately handled. If a 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. Figure 2 compares the packet processing time with and without handling the head-of-line blocking.

It is possible that a virus signature may be segmented into two contiguous data buffers. To avoid missing a match, when the virus scanning finishes, the tail data of $s$ characters will be moved to the prefix of the buffer, where $s$ is the maximum length of virus signatures. The subsequent data segments are appended to this segment in the buffer. A virus signature over two consecutive segments can therefore be detected.

### 3.1.6 Reconnection Problems

A connection cache is designed to identify a reconnection by keeping the information of a blocked connection, and to block this reconnection. Initially, 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 a connection is blocked, its source IP address, source port number, destination IP address, destination port number and protocol identifier will be stored into the connection cache. The packets having the same source IP address, destination IP address, destination port number and protocol identifier are viewed as in the reconnection, even though their source port numbers may be different. A P2P application, say BitTorrent, can switch to different source port numbers if a connection is blocked, so a connection with only a different source port is considered

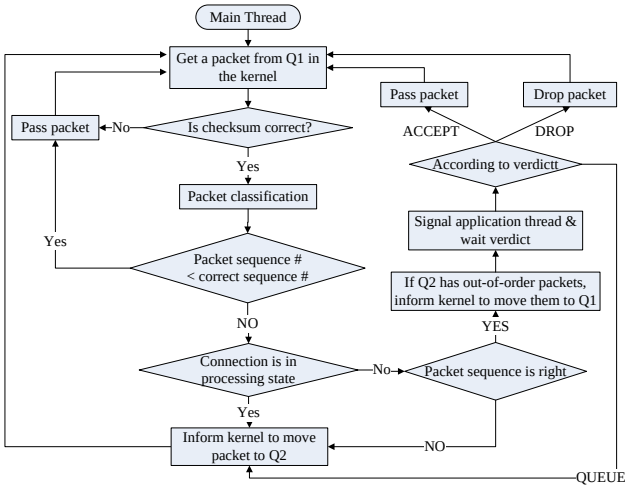


**Figure 3. The flow chart of the main thread.**

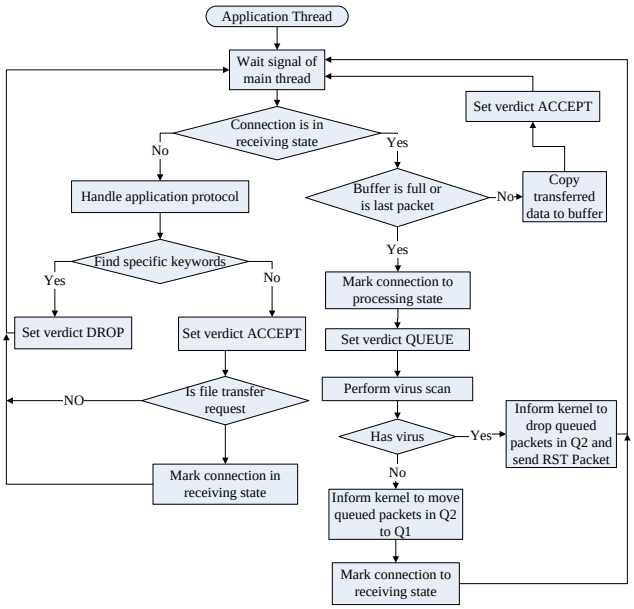


**Figure 4. The flow chart of an application thread.**

as a reconnection. A reconnection of a denied connection can be immediately dropped by the connection cache and thus the performance is enhanced.

#### 3.1.7 Out-of-order Packets

If a receiver sends three duplicated ACKs to its peer due to receiving out-of-order packets, the peer will assume packet loss and retransmit what appears to be a missing packet without waiting the timeout of the TCP retransmission timer. However, the retransmission will be redundant if it is not due to packet loss, but due to the queueing of out-of-order packets inside the gateway for packet reassembly and virus scanning. The redundant retransmission will decrease the throughput. This design duplicates out-of-order packets in the gateway and passes them immediately so that the receiver can respond ACKs as usual. Therefore, the redundant retransmission can be reduced.

### 3.2 System Implementation

The architecture design is based on the aforementioned ideas. The main thread in the kernel module gets packets from Q1 in the kernel and performs the pre-processing tasks. Because Q1 contains the packets from various connections, the kernel module uses the application number to identify the P2P protocol, and the main thread invokes an application thread to handle each connection related to that protocol. Figure 3 illustrates the entire flow of the main thread. After performing the pre-processing tasks, the main thread checks the connection state. If the connection is in the processing state, the main thread needs to handle the head-of-line blocking problem; otherwise, it signals the specific application thread to handle the packets.

Figure 4 illustrates the entire flow of an application thread, which handles a specific application protocol and decides to pass or drop the packets. If it needs to perform time-consuming content filtering or virus scanning, it marks this connection to be in the processing state and sets the verdict to `QUEUE`, and then the main thread can start to process subsequent packets. This approach can resolve head-of-line blocking.

Figure 5 illustrates and summarizes the operation of the proposed architecture, namely *kP2PADM*. The letter 'k' in the prefix of *kP2PADM* denotes the implementation in the kernel space. *kP2PADM* must occasionally call the *schedule* function in the Linux kernel to surrender the CPU control to other processes to avoid starvation. The CPU control will come back to *kP2PADM* if no other processes demand the CPU.

## 4 Performance Evaluation

### 4.1 Benchmarking Environment

We perform various benchmarks on *kP2PADM* installed on a PC with Pentium III 1 GHz CPU and 512 MB SDRAM. Two HTTP clients and three Web servers are in the test bed. Each client creates 100 threads for each server, and each thread downloads a 2 MB file from these three Web servers. Although the file size is small compared with some shared files, say movie files in real P2P applications, this

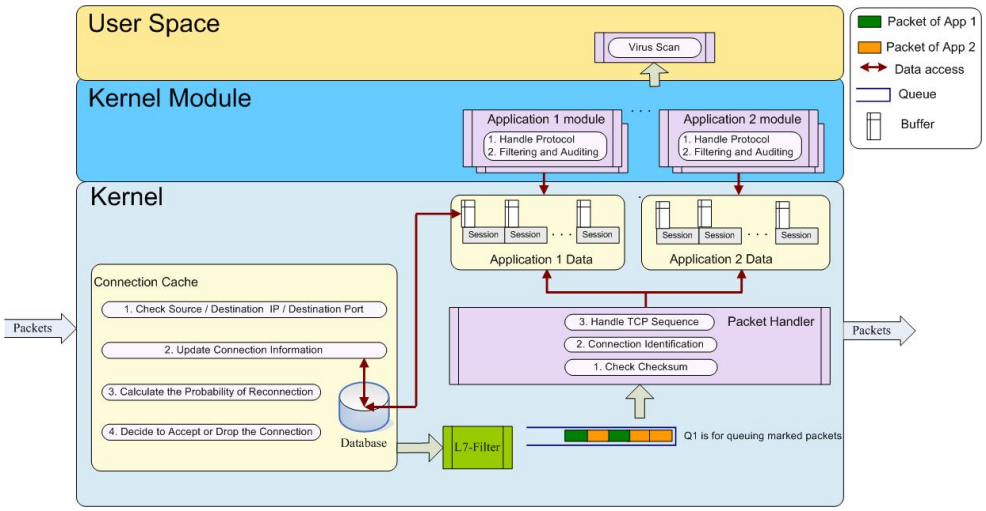


**Figure 5. The operation of the kP2PADM architecture.**

size is large enough to make file data dominate the P2P traffic, just like the case in real situations. We also implement a variant of this design in which the kernel module in *kP2PADM* is implemented as a daemon in the user space, namely *P2PADM*, to observe the performance gain of the in-kernel design compared with that of the in-daemon design.

We use HTTP traffic instead of real P2P traffic to benchmark *kP2PADM* for two reasons. (1) No such benchmark tools to date as we know generate P2P traffic for stress testing, and thus it is difficult to emulate a large amount of P2P traffic in a test bed. (2) Many P2P protocols, such as FastTrack and Gnutella, use HTTP-like protocol to transfer files. Although using HTTP traffic instead of P2P traffic is not the best choice, it is an acceptable choice given no appropriate tools available. The emulation can be similar to the case of file sharing because both contain mostly long packets. However, it is deviated from the cases of instant messages or queries for the location of files. For the latter cases, more studies should be conducted as future work.

## 4.2 Comparison with a Daemon Solution in the User Space

### 4.2.1 Throughput and CPU Utilization of kP2PADM

Throughput and CPU utilization are two important performance metrics of a gateway system. The following configurations are used to compare the impact of each function on performance. We use 'P2P proxy' as a generic term to mean the daemon in *P2PADM* and the kernel module in *kP2PADM* herein.

1. `Pure NAT`: the pure NAT function that only translates between private IP addresses and public IP addresses.
2. `NAT+packet queueing`: Besides NAT, every packet is queued in the kernel. *kP2PADM* just instructs the kernel to pass the packets without any further processing.
3. `NAT+packet queueing+L7`: Besides NAT plus packet queueing, the L7-filter is enabled with 20 rules. This process is similar to the former two, except that only HTTP is processed. This configuration is used to assess the performance impact from the L7-filter.
4. `P2P proxy+filtering`: All functions of P2P management are enabled except virus scanning and auditing. This configuration enables filtering transferred files according to the file name.
5. `P2P proxy+auditing`: The P2P proxy plus the auditing function on transferred files. It records the transferred files into the file system.
6. `P2P proxy+virus scanning`: The P2P proxy plus virus scanning on the transferred files.
7. `P2P proxy+filtering+auditing+virus scanning`: The P2P proxy plus all the above functions are enabled.

Figure 6 and 7 present the throughput and CPU utilization of both *P2PADM* and *kP2PADM* in each of the configurations.[1] Figure 7 presents not only the total CPU utilization but also the CPU utilization for the kernel. `Pure`

[1] We still encounter a bug of programming virus scanning in the kernel at the time of paper submission. The results related to virus scanning are estimated based on the amount of performance degradation in *P2PADM*.

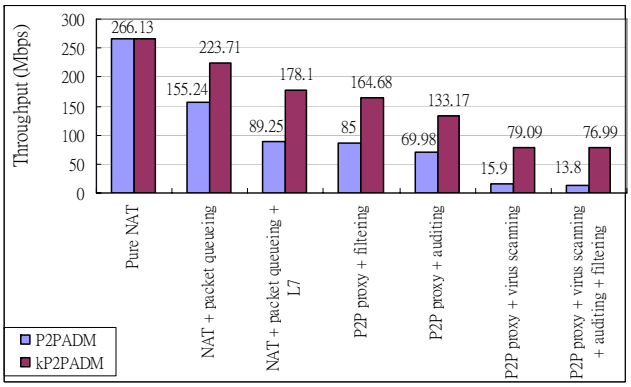


**Figure 6. Throughput of *P2PADM* and *kP2PADM*.**

NAT can reach the throughput about 266.13 Mbps on both *P2PADM* and *kP2PADM*. NAT+packet queueing reduces the throughput of *P2PADM* to 155.24 Mbps, but it reduces the throughput only slightly to 223.71 Mbps on *kP2PADM*. The latter is fast because the packets do not enter into the user space.

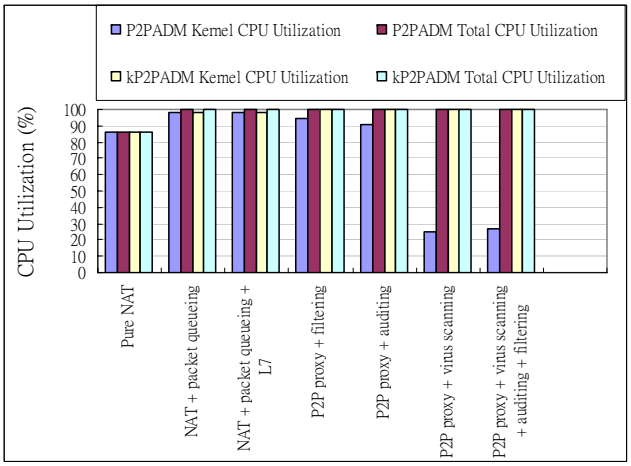


**Figure 7. CPU utilization of *P2PADM* and *kP2PADM*.**

If the L7-filter is enabled, the throughput decreases significantly to 89.25 Mbps on *P2PADM* and to 178.1 Mbps on *kP2PADM*. The influence on the throughput is moderate because the HTTP protocol is simple, but for more complex application protocols that need more processing, such as BASE64 encoding and decoding, we believe that the influence will be more significant. The influence of the auditing functions is also light. The throughput of P2P proxy+auditing on *P2PADM* is 69.98 Mbps and 133.17 Mbps on *kP2PADM*. *kP2PADM* always dominates

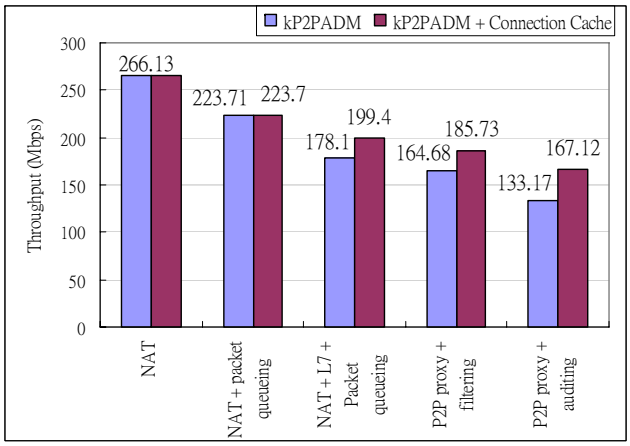


**Figure 8. Throughput of *kP2PADM* plus the connection cache.**

about 100% of CPU utilization while the P2P proxy is enabled. It is because *kP2PADM* is implemented in the kernel space and *kP2PADM* always occupies the CPU. If there are other processes to run, such an architecture should pay attention to surrendering the CPU to them in time, or the kernel will be blocked by *kP2PADM* for a long time.

### 4.3 Evaluation of the Connection Cache and Fast Pass

Figure 8 presents the throughput on *kP2PADM* as the connection cache is turned on. In the experiment, we set a policy on *kP2PADM* to block all packets from one of the two clients. This policy forces the blocked client to keep sending reconnection requests. The connection cache can increase the throughput by about 21 – 34 Mbps in the latter three configurations in Figure 8.

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) [2] on Linux. NIST Net allows a single Linux PC to act as a gateway to emulate a wide variety of network conditions, such as packet loss, out-of-order packets, transmission delay and so on. The traffic between peers passes through the NIST emulator besides the *kP2PADM* gateway to experience the emulated packet loss and delay. A 300 MB file is transmitted from one peer to the other in this benchmark.

Figure 9 presents the transfer time with and without fast pass for different packet loss rates. The packet loss rates range from 0% to 5% to emulate the cases in a real environment [9]. Fast pass can shorten the transfer time between two peers. Two observations are in the experimental results: (1) the higher the packet loss rate is, the more the transfer time can be shortened with fast pass, and (2) the longer the

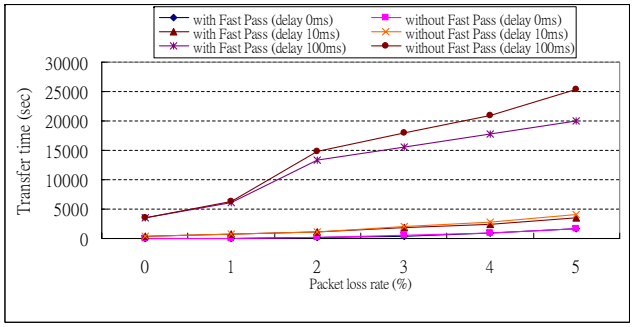


**Figure 9. Transfer time with and without fast pass for different packet loss rates.**

delay is, the more the transfer time can be shortened. Both can be justified by the reason that the queueing time in the gateway is much longer with higher packet loss rate and longer delay, so the transfer time can be shortened more.

### 4.4 Internal Benchmarking

To further identify the improvements and the bottlenecks of *kP2PADM*, we examine the execution time of each stage in the entire packet processing flow with all the functions turned on. The execution time is measured by calculating the difference of timestamps taken from the `do_gettimeofday()` kernel function, which supports resolution up to $\mu$s, in the beginning and the end of a code segment. Table 3 presents the internal benchmarking results of *P2PADM* and *kP2PADM*. Moving the code from the user space to the kernel space can reduce the execution time in most of the stages, especially those of getting packets for processing, handling TCP sequence and auditing. The improvement of these three stages are the most significant because they heavily depend on moving the packets between the kernel space and the user space. Handling the HTTP protocol should have shown significant improvement, but it does not because processing the HTTP protocol takes longer time than data passing between the kernel space and the user space.

## 5 Conclusions

This work presents an in-kernel gateway design with four major functions for P2P management. Several practical techniques are proposed to enhance the system performance. The dual-queue architecture can effectively eliminate head-of-line blocking. Besides, *kP2PADM* also resolves the possible performance degradation from useless reconnection requests and out-of-order packets. Through the connection cache and fast pass, we can increase the

| Stage | *P2PADM* (ms) | *kP2PADM* (ms) |
|---|---|---|
| Getting packets | 30 | 5 |
| Checking the checksum | 8 | 7 |
| Packet classification | 5 | 5 |
| Handling TCP sequence | 28 | 10 |
| Handling HTTP | 30 | 28 |
| Auditing | 30 | 12 |
| Setting the verdict | 7 | 6 |

**Table 3. Execution time of each stage in the internal benchmark.**

throughput of *kP2PADM* and reduce the transmission time. The in-kernel design is also compared with an in-daemon design in terms of the throughput and CPU utilization to show the benefits of implementation in the kernel.

The external benchmark results indicate that in-kernel management improves the performance of the daemon version of *P2PADM*. The throughput can achieve 185.73 Mbps from 85 Mbps in the functions of proxy and content filtering. The throughput is more than double that of *P2PADM*. The connection cache can increase the throughput by about 21 – 34 Mbps. The CPU utilization of *kP2PADM* always reaches 100% because all the functions of *kP2PADM* are implemented in the kernel space. Fast pass can reduce more transfer time when the delay time is longer or the packet loss rate is larger.

## References

[1] S. A. Baset and H. Schulzrinne. An analysis of the Skype peer-to-peer internet telephony protocol. In *Proceedings of IEEE INFOCOM*, Apr. 2006.

[2] M. Carson and D. Santay. NIST Net - a Linux-based network emulation tool. 2003.

[3] T. Karagiannis, A. Broido, N. Brownlee, k. claffy, and M. Faloutsos. Is P2P dying or just hiding? In *Proceedings of Globecom*, Nov. 2004.

[4] T. Karagiannis, A. Broido, M. Faloutsos, and k. claffy. Transport layer identification of P2P traffic. In *ACM. SIGCOMM/USENIX Internet Measurement Conference (IMC)*, Oct. 2004.

[5] A. Parker. P2P in 2005. Available at `http://www.cachelogic.com/home/pages/studies/2005_01.php`.

[6] S. Sen, O. Spatscheck, and D. Wang. Accurate, scalable in-network identification of p2p traffic using application signatures. In *Proceedings of International WWW Conference*, May 2004.

[7] S. Sen and J. Wang. Analyzing peer-to-peer traffic across large networks. *IEEE/ACM Transactions on Networking*, 12(2):219–232, Apr. 2004.

[8] A. Spognardi, A. Lucarelli, and R. D. Pietro. A methodology for p2p file-sharing traffic detection. In *Proceedings International Workshop on Hot Topics in Peer-to-Peer Systems*, 2005.

[9] M. Uajnik, S. Moon, J. Kurose, and D. Towsley. Measurement and modeling of the temporal dependence in packet loss. *Tech. Rep. UMASS CMPSCI 98-78*, 1998.