IEICE TRANS. INF. & SYST., VOL.E91–D, NO.10 OCTOBER 2008

PAPER

kP2PADM: An In-Kernel Architecture of P2P Management Gateway*

Ying-Dar LIN[†a)], ***Member***, **Po-Ching LIN**[††], **Meng-Fu TSAI**[††], **Tsao-Jiang CHANG**[††], ***and*** **Yuan-Cheng LAI**[†††], ***Nonmembers***

SUMMARY Managing increasing traffic from Instant Messengers and P2P applications is becoming more important nowadays. We present an in-kernel architecture of management gateway, namely *kP2PADM*, built upon open-source packages with several modifications and design techniques. First, the in-kernel design streamlines the data path through the gateway. Second, the dual-queue buffer eliminates head-of-line blocking for multiple connections. Third, a connection cache reduces useless reconnection attempts from the peers. Fourth, a fast-pass mechanism avoids slowing down the TCP transmission. The in-kernel design approximately doubles the throughput of the design in the user space. The internal benchmarks also analyze the impact of each function on performance.

key words: *P2P/IM management, gateway, in-kernel implementation*

1. Introduction

The traffic of peer-to-peer (P2P) applications has dominated the Internet traffic lately [1], [2]. Properly managing P2P traffic is therefore important. For example, Internet service providers can restrict P2P traffic to avoid excessive occupation of bandwidth. Restricting the use of instant messengers (IM) can increase employee productivity and prevent confidential information from leakage. Like firewalls, which control accesses from and to the intranet, a transparent gateway that centrally manages P2P traffic is a promising approach.

Managing P2P traffic** is more complicated than managing other traffic. Conventional approaches that identify applications according to fixed port numbers no longer work because most P2P applications run on dynamic ports and tend to hide themselves by encrypting the messages [3]. The authors in [4], [5] indicate the insufficiency of detecting P2P traffic with port numbers. They analyze the P2P traffic and extract signatures from it for high detection accuracy. Some methods do not rely on signatures, but on characteristics of P2P traffic such as connection patterns [2], [6] because the messages may be encrypted. These designs focus on increasing the accuracy, but P2P gateway design has practical issues other than accuracy. For example, packets should be queued and reassembled in the gateway, and wait the result of deep content inspection for virus signatures or certain sensitive keywords before further processing. Efficient queue management and streamlined packet flow are therefore essential to a scalable design. Besides filtering P2P traffic, a fair amount of research work studies how to accelerate analyzing and filtering on network stream information, we refer the readers to [7] on the issues. This work relates the acceleration from the system architecture, which is not addressed in that research work.

This work designs an in-kernel architecture on Linux, namely *kP2PADM*, to manage P2P traffic on a transparent gateway, where '*k*' emphasizes the in-kernel design and *ADM* stands for administration. Several management functions are implemented: (1) classifying and filtering P2P traffic, (2) scanning shared files for viruses, (3) auditing chatting messages and transferred files and (4) bandwidth control. The design streamlines the data path through the gateway and reduces the overheads of passing packets between the kernel space and the user space. The L7-filter (`l7-filter.sourceforge.net`) acts as the connection classifier to identify P2P signatures in the application-layer messages. If the packets belong to P2P connections, they are redirected to a kernel module for further processing such as packet reassembly and content filtering; otherwise, they are simply passed through the gateway. The kernel module can ban undesirable applications or perform bandwidth control according to the management policy.

The module implements TCP reassembly to handle raw packets that may be out-of-order, lost or duplicated. After reassembly, the packet content is ready for content filtering or virus scanning. The 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 processing. We propose a dual-queue mechanism for this situation, and modify a queue handler *ip_queue* in the Linux source to manage 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 individual application protocols and perform the desired filtering.

This work also addresses two negative factors that

Manuscript received December 10, 2007.
Manuscript revised April 12, 2008.

[†]The authors is with the Department of Computer Science, National Chiao Tung University, Taiwan.
[††]The authors are with High-speed Lab, Department of Computer Science, National Chiao Tung University, Taiwan.
[†††]The author is with the Department of Information Management, National Taiwan University of Science and Technology, Taiwan.

*This work was supported in part by the Taiwan National Science Council's Program of Excellence in Research, and in part by grants from Cisco and Intel.

a) E-mail: ydlin@cis.nctu.edu.tw

DOI: 10.1093/ietisy/e91–d.10.2398

**Because IM applications operate in a P2P mode, we use the term P2P when referring to P2P/IM for simplicity.

could reduce the performance: useless reconnection requests from P2P applications and out-of-order packets. Some users or P2P applications persistently attempt to reconnect to their peers within a short period of time when the gateway blocks the connection establishment. Handling these attempts is wasteful, and they should be blocked again soon. This architecture deploys a connection cache to block reconnection requests as soon as possible. Out-of-order packets result in unnecessary packet retransmission because the gateway must queue them to maintain the order. The sender may consider the queueing as a sign of packet loss if its TCP retransmission timer expires or it receives three duplicated TCP ACKs from the peer. We use a mechanism called *fast pass* that passes out-of-order packets immediately, and duplicates a copy of each packet in the kernel for ordering and reassembly.

This work proposes several strategies to resolve practical design issues of an efficient P2P management gateway. The rest of this work is organized as follows. Section 2 overviews typical P2P applications and surveys related packages and products. Section 3 presents the key ideas of the system architecture and details the implementation. Section 4 presents the performance evaluation of this system and analyzes the result. Section 5 concludes the study and indicates future work.

2. Survey of Related Works

2.1 Overview of P2P Applications

Table 1 summarizes the characteristics of popular P2P and IM applications: Kazaa (`www.kazaa.com`), eMule (`www.emule-project.net`), BitTorrent (`www.bittorrent.com`), Gnutella (`www.gnutella.com`), MSN Messenger (`messenger.msn.com`) and Skype (`www.skype.com`). Most P2P applications set the default port numbers upon installation, but the actual port numbers in use may be either changed by the users later or determined by the applications. P2P applications tend to circumvent the filtering of firewalls by running on dynamic ports or hiding themselves on well-known ports. An example is Skype, which can choose the ports of HTTP and HTTPS in case of connection failure [8]. Kazaa can hop to another port if it is blocked in a port. Generally, it is impossible to tell the application from only the port number. Therefore, common solutions to identify P2P applications are based on signatures or heuristics on the connection patterns.

P2P applications may sequentially transfer a file from another peer, or in out-of-order segments. If a file is either encrypted or transferred out-of-order, performing virus scanning and auditing becomes intractable. For example, a user may download a file with a laptop in two different locations. The management gateway in either location sees only random fragments that are available. Only the laptop itself can reassemble and restore the original content. Therefore, we do not support both functions for 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 and block the file transfer. Moreover, enterprises may not want employees to leak out confidential information via chatting messages. Filtering sensitive keywords or recording the message is also needed. The *kP2PADM* architecture is designed to implement feasible management functions summarized in Table 2.

Table 1 The characteristics of P2P and IM applications. ST=sequential transfer, EN=with encryption, DP=dynamic port, FV=filename visibility, DFP=default ports.

applications	ST	EN	DP	FV	DFP
Kazaa	Yes	Yes	No	Maybe	1214
eMule	No	No	No	No	4661–4665
BitTorrent	No	No	No	Yes	6881–6889
Gnutella	Yes	No	No	Yes	6346, 6347
MSN Messenger	Yes	No	No	Yes	1863, 6891–6900, 6901
Skype	Yes	Yes	Yes	No	randomly chosen

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

applications	CF	FT	VS	AU	BC
Kazaa	Yes	Yes	Yes	Yes	Yes
eMule	Yes	Yes	No	No	Yes
BitTorrent	Yes	Yes	No	No	Yes
Gnutella	Yes	Yes	Yes	Yes	Yes
MSN	Yes	Yes	Yes	Yes	Yes
Skype	Yes	No	No	No	Yes

2.2 Related Products and Implementations

P2P and IM traffic management has attracted much attention recently. We are aware of several commercial machines for the demand, such as Facetime (`www.facetime.com`) and Akonix (`www.akonix.com`), to name a few, as well as software tools such as p2pwatchdog (`www.p2pwatchdog.com`) and TerminatorX (`www.plevna.f9.co.uk`). These products can manage P2P/IM traffic by filtering and auditing messages on IM, block P2P traffic, detect malware, and so on. However, these products reveal little about the design due to their black-box nature, let alone evaluation of the internal functions.

This work prefers to integrate open-source packages into the kP2PADM management gateway. Packages such as L7-filter and IPP2P (`www.ipp2p.org`) are classifiers that identify P2P traffic by inspecting the packet payload in the Linux Netfilter subsystem (`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 contain the signatures, the module marks the entire connection as identified. IPP2P checks every packet in a connection. It embeds the signatures in the source 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 integrates the L7-filter as the classifier. Note that although it is possible to use Netfilter alone to perform basic shaping, classifying and filtering, the kP2PADM gateway can do more beyond that.

Implementation in the kernel space can reduce moving data between the kernel/user space and increase the system performance. The authors in [9] presented the connection-splicing technique to speed up data forwarding in the kernel level. The idea of in-kernel implementation is also applied to a Web proxy and a Web switch to improve the efficiency [10], [11]. A Web Server named kHTTPd (`www.fenrus.demon.nl`) running as a Linux module also shows higher performance than user-space Web servers of Apache and Zeus. Their success tantalized us to implement an in-kernel design of the management gateway. In Sect. 4.2, we will compare the performance of the in-kernel design and the in-daemon design that runs in the user space.

3. Design of System Architecture

P2P connection classification involves examining application messages, which are available only after a TCP connection between two peers is established. Because the gateway transparently monitors the traffic, a peer does not have to establish a connection with the gateway. When the gateway monitors and discovers a new connection, it redirects the connection from the kernel to the kernel module for the following processing.

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 for signature matching. A predefined protocol number that indicates the traffic type is assigned to a connection identified as P2P traffic. The kernel then filters undesirable traffic and controls the available bandwidth according to this number. Note that the packets containing important information such as the file name or size might have been already passed to the peer before the traffic type is identified. Because the kernel module may still need such information to take the actions, the collected application data are packed into a special packet created inside the kernel for each connection. This packet is only internally passed to the kernel module for extracting the application data and further processing.

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 the unidentified packets are just passed through the gateway. The queued packets (actually, only the pointers to the packets for efficiency) are passed to the kernel module, which processes these packets and sequentially sets the verdict on them. The verdict includes

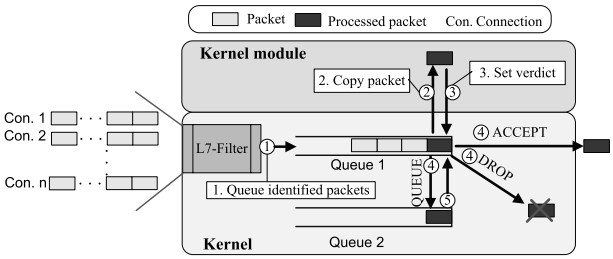

Fig. 1 Packet queueing and redirecting mechanism.

`ACCEPT`, `DROP` or `QUEUE`. `ACCEPT` means passing a packet to the peer, while `DROP` means dropping a packet. If a packet cannot be decided to be passed or dropped at the present time (e.g., waiting for virus scanning), the verdict will be `QUEUE`, and the packet will be moved from Q1 to Q2 to wait temporarily[†]. Figure 1 illustrates the operation.

3.1.3 Packet Preprocessing

When the kernel module gets a packet from Q1, it performs three steps before handling the P2P protocol: (1) The packet checksum (IP/TCP/UDP checksum, if applicable) is examined. The module simply drops the packet with an incorrect checksum. (2) Packet classification identifies which connection the packet belongs to. (3) The correct sequence of the packet is handled before reassembly.

Packets in the marked connections are further classified based on the five tuples: 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, which can reorder the packets in the correct sequence. The kernel module therefore compares the sequence number of the handled packet with the correct one. If the number is smaller, the packet is a duplicated one and should be dropped. If the number is larger, the packet should wait for the presence of missing packets. The kernel will temporarily move this packet from Q1 to Q2 for the waiting. If the sequence number is correct, the out-of-order packets in Q2, if any, will be moved back to Q1. The kernel module then reassembles these packets. To prevent a denial-of-service attack, the out-of-order packets in Q2 should be cleared out after a timeout of not being reassembled.

3.1.4 P2P Protocol Processing

The P2P protocol is recognized according to the assigned protocol number, and handled after packet reassembly. 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

[†]The other situation to move packets from Q1 to Q2 is when the packet sequence is incorrect and the connection is in the processing state. See Fig. 3 for the flow chart.

data.

The packets are checked based on the application protocol to examine whether a chatting message or a file request contains specific keywords. If a 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 will pass the packet and record the chatting message or file name in an external storage if auditing is enabled. Bandwidth control is in this stage by limiting the bandwidth occupied by a connection. If the packet should not be blocked, the connection is marked to be in the receiving state. The subsequent data segments are then reassembled, and virus scanning is performed if needed.

3.1.5 Virus Scanning for Shared File

A buffer is allocated for receiving a connection to be scanned for viruses. When the kernel module receives a packet, it first checks whether the buffer is full or whether the packet is the last one in the transferred file. If either happens, a virus scanning program (We use the code from ClamAV (`www.clamav.net`) in the user space performs virus scanning on the buffer. If a virus is found, the program tells the kernel to drop the packets to destroy the file, and sends an RST packet to the peer to tear down the connection; otherwise, the packets are passed.

The above method may have two problems: *head-of-line blocking* and segmentation of virus signatures. First, 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. Because this will constrain the throughput of the entire system, the connection to be scanned is marked as in the processing state, and another thread is called to perform virus scanning. The subsequent packets of the same connection will be moved from Q1 to Q2, so the packets in other connections can be immediately handled. If a virus is found, the packets in Q2 associated with this connection will be dropped; otherwise, they are moved back from Q2 to Q1. Figure 2 compares packet processing with and without handling the head-of-line blocking. Without head-of-line blocking, processing the packets in other connections can be interleaved with virus scanning. Second, a virus signature may span two consecutive buffers, so it will not be found in either buffer. To avoid missing a match, after a buffer has been scanned, the buffer tail of $s-1$ characters will be moved to the prefix of the next buffer, where s is the maximum length of virus signatures. The signature therefore can be found in the next buffer.

3.1.6 Reconnection Problems

A connection cache can keep the information of a blocked connection and recognize its 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 four tuples of source IP address, destination IP address, destination port number and protocol identifier will be recorded in the connection cache. The subsequent packets having the same four tuples are viewed as in the reconnection, even though their source port numbers are different. The source port number is not counted because P2P applications, say BitTorrent, may switch to different source port numbers when their connections are blocked. The connection cache will immediately drop a reconnection to increase the performance.

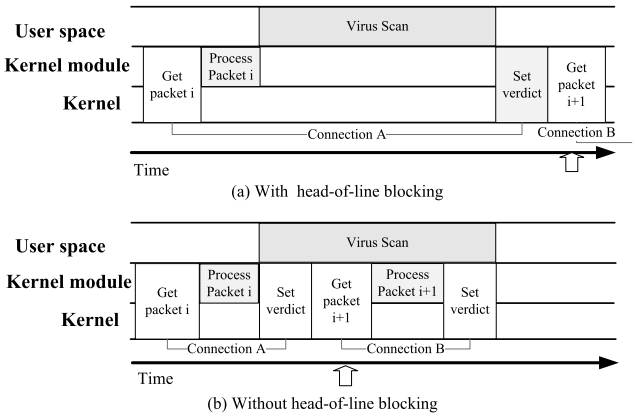

Fig. 2 Comparison of packet processing with and without handling the head-of-line blocking. The latter can concurrently scan viruses and process the packets in other connections.

3.1.7 Fast Pass for Out-of-Order Packets

If an packet is queued in the gateway for packet reordering, a sender may view the out-of-order packet as being lost, and retransmit it or enter the slow-start process. However, the retransmission is redundant or the slow-start is unnecessary because it is not due to packet loss, but due to the queueing of out-of-order packets in the gateway. The redundant retransmission will decrease the throughput. This design prefers to immediately pass out-of-order packets, so that the receiver can soon respond with ACKs to reduce the redundant retransmission and avoid slow-start as usual. A copy of each out-of-order packet is kept in the gateway for reassembly and examination. This mechanism is called fast pass in the design.

3.2 System Implementation

The system is multi-threaded. The main thread in the kernel module gets packets from Q1 in the kernel and performs the pre-processing tasks. Because Q1 contains 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 connec-

tion is in the processing state, the main thread handles the head-of-line blocking problem; otherwise, it signals a 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. When the application thread performs time-consuming content filtering or virus scanning, it marks the connection to be in the processing state and sets the verdict to `QUEUE`. The main thread can start to process subsequent packets. This approach resolves head-of-line blocking.

Figure 5 illustrates and summarizes the operation of the *kP2PADM* architecture. *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.

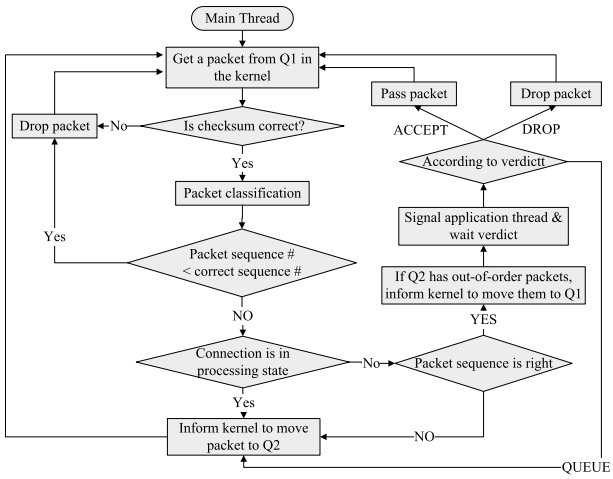

Fig. 3 The flow chart of the main thread.

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. Figure 6 presents the topology of the testing environment. 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 the three Web servers. Although the file size is small compared with some shared

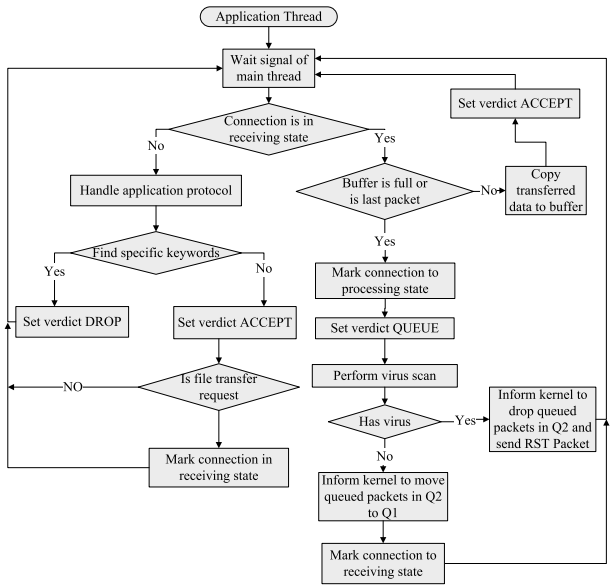

Fig. 4 The flow chart of an application thread.

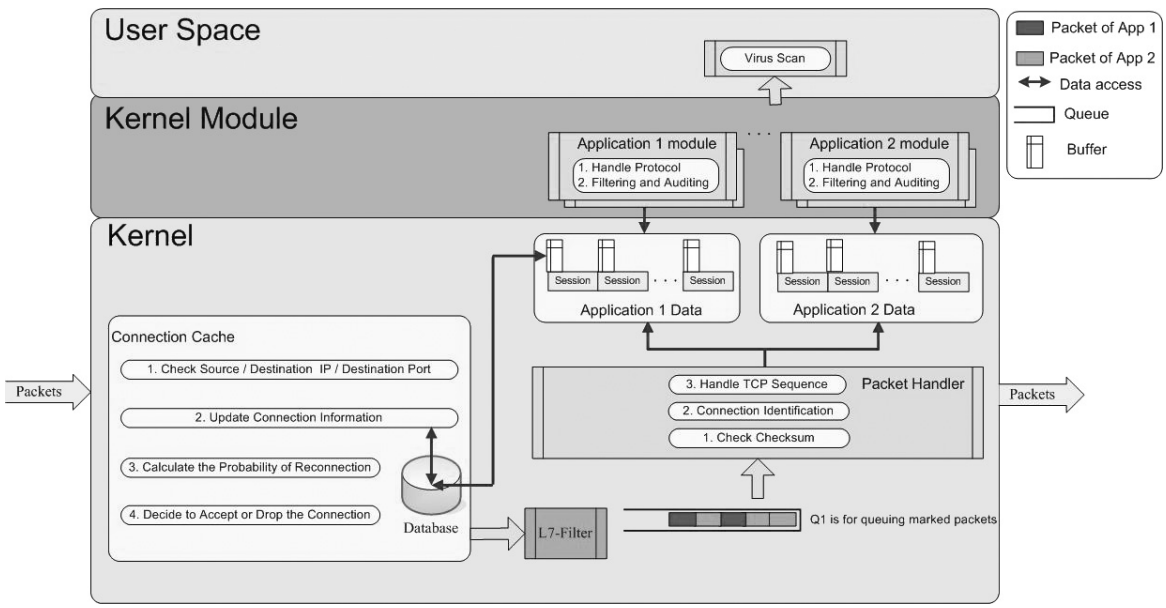

Fig. 5 The operation of the kP2PADM architecture.

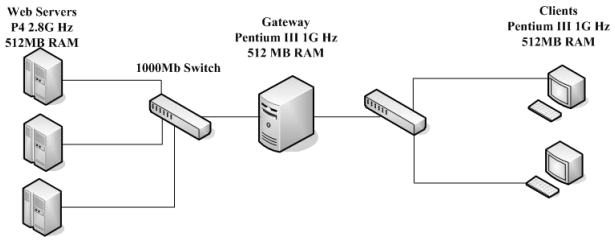

Fig. 6 The topology of the testing environment.

files, say movie files in real P2P applications, this size is large enough to make file data dominate the traffic, just like the practical case. We also implement a variant of this design in which the kernel module in *kP2PADM* is implemented as a user-space daemon, namely *P2PADM*, to compare the performance of the in-kernel and the in-daemon design.

We use HTTP traffic instead of real P2P traffic to benchmark *kP2PADM* for two reasons. (1) No appropriate benchmark tools to date as we know generate P2P traffic for stress testing, so 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 acceptable in terms of performance evaluation without appropriate stress-testing tools. The emulation is similar to file sharing because both contain mostly long packets, but it is deviated from the cases of instant messages or queries for the location of files. The latter cases deserves further study in the future.

4.2 Comparison with a User-Space Daemon Solution

4.2.1 Throughput and CPU Utilization of kP2PADM

We use the following configurations to compare the impact of each function on the throughput and CPU utilization. 'P2P proxy' denotes a generic term for the daemon in *P2PADM* and the kernel module in *kP2PADM*.

1. `Pure NAT`: the pure NAT function only translates between private 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 the above two, the L7-filter is enabled with 20 rules. This configuration assesses 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, and records the 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 with all the above functions enabled.

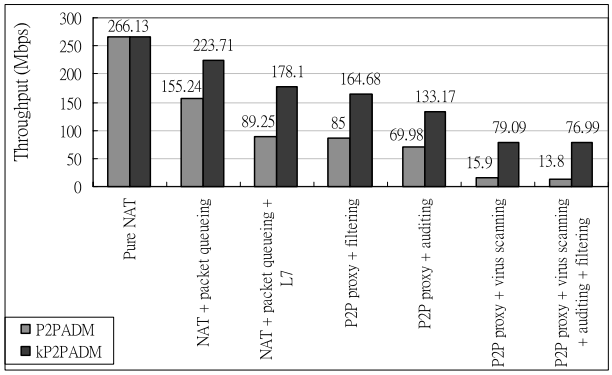

Fig. 7 Throughput of *P2PADM* and *kP2PADM*.

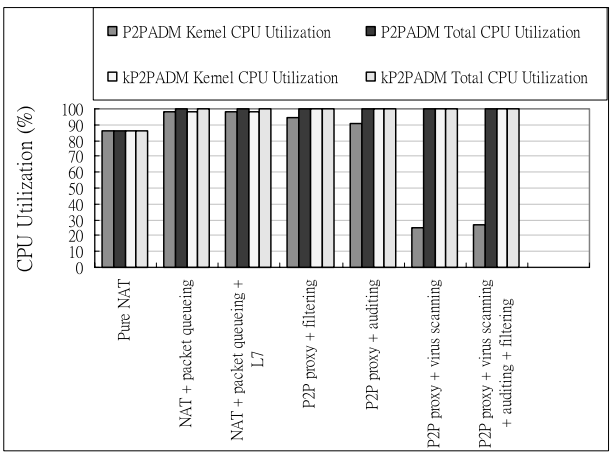

Fig. 8 CPU utilization of *P2PADM* and *kP2PADM*.

Figures 7 and 8 present the throughput and CPU utilization of both *P2PADM* and *kP2PADM* in each of the configurations[†]. Figure 8 presents not only the total CPU utilization but also the CPU utilization for the kernel. `Pure 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.

If the L7-filter is enabled, the throughput decreases significantly to 89.25 Mbps on *P2PADM* and to 178.1 Mbps on *kP2PADM*. The degradation primarily comes from signature matching in the L7-filter. Another major degradation comes from virus scanning, which involves string matching against a large signature set. The observation that string matching is a bottleneck is consistent with that

[†]We encountered a bug of programming virus scanning in the kernel, and the results related to virus scanning are estimated based on the amount of performance degradation in *P2PADM*.

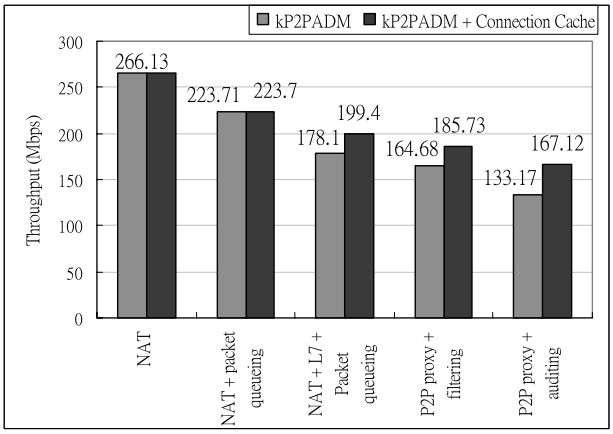

Fig. 9 Throughput of *kP2PADM* plus the connection cache.

in other fields, such as intrusion detection. The influence of the auditing functions is light. The throughput of `P2P proxy+auditing` on *P2PADM* is 69.98 Mbps and 133.17 Mbps on *kP2PADM*. Note that *kP2PADM* always dominates nearly 100% of CPU utilization beyond the pure NAT function because *kP2PADM* is implemented in the kernel space and *kP2PADM* always occupies the CPU. If there are other processes to run, this architecture should pay attention to surrendering the CPU to them in time, or *kP2PADM* will block the kernel for a long time.

4.3 Evaluation of the Connection Cache and Fast Pass

Figure 9 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 around 21 $\sim$ 34 Mbps in the latter three configurations.

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) [12] 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 10 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 [13]. 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 fast pass can shorten the transfer time, and (2) the longer the delay is, the more the transfer time can be shortened. Both can be justified because the queueing time in the gateway is much longer with higher packet loss rate and longer delay. Note

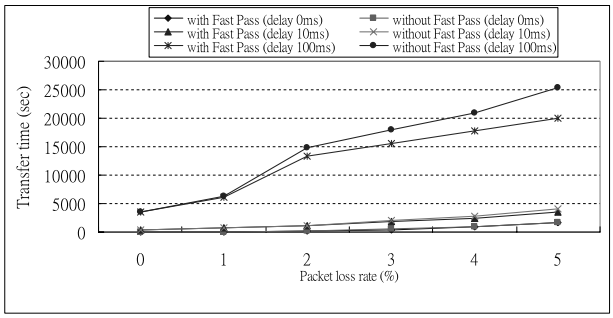

Fig. 10 Transfer time with and without fast pass for different packet loss rates.

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

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

that the shortening is not obvious with short delay and low packet loss rate in Fig. 10, and the lines with and without fast pass almost overlap with each other.

4.4 Internal Benchmarking

To further identify the improvement and the bottleneck 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 μs, 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 application protocol should have shown significant improvement, but it does not because application protocol processing (e.g., content filtering) is itself time-consuming and in-kernel processing helps little in this respect.

5. Conclusions and Future Work

This work presents an in-kernel gateway design from open-source packages for P2P management. Despite the existence of some commercial products, they are black-boxes without detailed accounts of how to build a P2P management gateway. This paper relates experiences in building such a system, and contributes detailed descriptions of several practi-

cal techniques to enhance the performance. The dual-queue architecture can effectively eliminate head-of-line blocking. Moreover, *kP2PADM* also resolves the possible performance degradation from useless reconnection requests and out-of-order packets with the connection cache and fast pass, respectively. The external benchmark indicates that in-kernel management improves the performance of the daemon version of *P2PADM*. The throughput is nearly doubled with respect to the functions of proxy and content filtering, from 85 Mbps in *P2PADM* to 164.68 Mbps. The connection cache can further increase the throughput by around 21 ∼ 34 Mbps. Fast pass can reduce more transfer time when the delay time is longer or the packet loss rate is higher.

Some applications such as Skype encrypt transmitted messages. Because the gateway acts as an intermediary, it is intractable to tap the messages without the breaking the key. It is a fundamental issue in cryptography. Therefore, dealing with information leakage is intractable except blindly blocking all message transmission. We believe a general solution to this problem deserves further study in future research.

References

[1] A. Parker, P2P in 2005. http://www.cachelogic.com/home/pages/studies/2005_01.php
[2] S. Sen and J. Wang, "Analyzing peer-to-peer traffic across large networks," IEEE/ACM Trans. Netw., vol.12, no.2, pp.219–232, April 2004.
[3] T. Karagiannis, A. Broido, N. Brownlee, K. Claffy, and M. Faloutsos, "Is P2P dying or just hiding?," Proc. Globecom, Nov. 2004.
[4] S. Sen, O. Spatscheck, and D. Wang, "Accurate, scalable in-network identification of P2P traffic using application signatures," Proc. International WWW Conference, May 2004.
[5] A. Spognardi, A. Lucarelli, and R.D. Pietro, "A methodology for P2P file-sharing traffic detection," Proc. International Workshop on Hot Topics in Peer-to-Peer Systems, 2005.
[6] T. Karagiannis, A. Broido, M. Faloutsos, and K. Claffy, "Transport layer identification of P2P traffic," ACM SIGCOMM/USENIX Internet Measurement Conference (IMC), Oct. 2004.
[7] P.C. Lin, Y.D. Lin, Y.C. Lai, and T.H. Lee, "Using string matching for deep packet inspection," Computer, vol.41, no.4, pp.23–28, April 2008.
[8] S.A. Baset and H. Schulzrinne, "An analysis of the Skype peer-to-peer internet telephony protocol," Proc. IEEE INFOCOM, April 2006.
[9] O. Spatscheck, J.S. Hansen, J.H. Hartman, and L.L. Peterson, "Optimizing TCP forwarder performance," IEEE/ACM Trans. Netw., vol.8, no.2, pp.146–157, April 2000.
[10] J.L. Zhou, J.F. Yu, and H.T. Xia, "Data stream splicing for Web proxy cache optimization," Frontier of Computer Science and Technology (FCST), Nov. 2006.
[11] Y.K. Chang, W.H. Cheng, and C.P. Young, "Fully pre-splicing TCP for Web switches," Proc. Innovative Computing, Information and Control (ICICIC), Aug. 2006.
[12] M. Carson and D. Santay, NIST Net — A Linux-based network emulation tool, 2003.
[13] 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.

Ying-Dar Lin received the bachelor's degree in Computer Science and Information Engineering from National Taiwan University in 1988, and the M.S. and Ph.D. degrees in Computer Science from the University of California, Los Angeles in 1990 and 1993. He joined the faculty of the Department of Computer and Information Science since 1993. From 2005, he is the director of the graduate Institute of Network Engineering. He is also the founder and director of Network Benchmarking Lab since 2002. His research interests include design, analysis, implementation and benchmarking of network protocols and algorithms, wire-speed switching and routing, quality of services, network security, content networking, network processors and SoCs, and embedded hardware software co-design.

Po-Ching Lin received the bachelor's degree in Computer and Information Education from National Taiwan Normal University, Taipei, Taiwan in 1995, and the M.S. degree in Computer Science from National Chiao Tung University, Hsinchu, Taiwan in 2001. He is a Ph.D. candidate of Computer Science in National Chiao Tung University. His research interests include content networking, algorithm designing and embedded hardware software co-design.

Meng-Fu Tsai received the bachelor's degree and the M.S. degree in Computer Science from National Chiao Tung University, Hsinchu, Taiwan in 2003 and 2005. He is a software engineer in ZyXEL since 2005. His research interests include network security and content networking.

Tsao-Jiang Chang received the bachelor's degree and the M.S. degree in Computer Science from National Chiao Tung University, Hsinchu, Taiwan in 2004 and 2006. His research interests include P2P Gateway and performance evaluation.

Yuan-Cheng Lai received the bachelor's degree and the M.S. degree in Computer Science and Information Engineering from National Taiwan University in 1988 and 1990, and the Ph.D. degree from Computer and Information Science, National Chiao Tung University in 1997. He joined the faculty of National Cheng-Kung University, Tainan, Taiwan in 1998. He is an associate professor in Department of Information Management, National Taiwan University of Science and Technology, Taipei, Taiwan. His research interests include high-speed networking, wireless network and network performance evaluation, Internet applications.