国立交通大學

資訊科學系

碩 士 論 文

以串流為基礎具有交錯解壓縮與病毒掃瞄的郵件代理伺服器

A Stream-based Mail Proxy with Interleaved Decompression and Virus Scanning

研 究 生：陳思豪

指導教授：林盈達 教授

中 華 民 國 九 十 四 年 六 月

以串流為基礎具有交錯解壓縮與病毒掃瞄的郵件代理伺服器

A Stream-based Mail Proxy with Interleaved Decompression and Virus Scanning

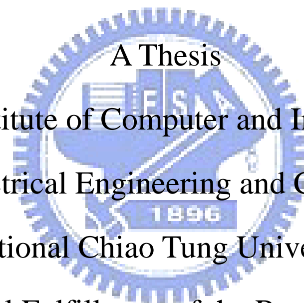
研 究 生： 陳思豪 Student : Szu-Hao Chen

指導教授： 林盈達 Advisor : Ying-Dar Lin

國 立 交 通 大 學

資 訊 科 學 研 究 所

碩 士 論 文



A Thesis

Submitted to Institute of Computer and Information Science

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer and Information Science

June 2005

HsinChu, Taiwan, Republic of China

中華民國九十四年六月

以串流為基礎具有交錯解壓縮
與病毒掃瞄的郵件代理伺服器

學生 ： 陳思豪　　　　　　指導教授 :林盈達

國立交通大學資訊科學系

摘要

在閘道器或防火牆系統上防毒時有中央控管與提早擋下病毒等優點。但管理一群電腦時，傳統的先存下整個資料再處理的方法會有資源耗損太快的問題以及大量的檔案系統存取負荷。我們實作了一個以串流為基礎的郵件代理伺服器，它以交錯執行分析 MIME、解碼、解壓縮、掃毒等步驟達到部分地處理郵件而不是先將整封存起來。在實作上，我們整合了一些開放源碼的套件，並且使用系統呼叫 select 將其實作成單一程序的多工伺服器。這個系統完全沒有存取檔案系統時的負荷，並且使用較少量的記憶體。我們的評測程式說明了在許多種的郵件上，我們的代理伺服器與先存檔再處理的代理伺服器(以 AMaViS 和 postfix 兩套件組成)比起來同時具有更好的速度與更少的系統資源使用率。在測試數據中我們發現我們的代理伺服器在沒有任何處理單純轉送封包的情況下比傳統儲存全部的方法快七倍；在有掃毒的情況下快三倍；在有掃毒且有解壓縮的情況下快兩倍。我們的系統在記憶體的使用上，不論該連線所傳送的資料大小，對單一連線皆維持一個定值，總使用量隨著連線數線性成長；但傳統的方法在儲存空間上與連線數與資料大小皆成正比。

關鍵字: 串流，分段，線上，即時，病毒，掃毒，代理伺服器，交錯，解壓縮

A Stream-based Mail Proxy with Interleaved Decompression and Virus Scanning

Student: Szu-Hao Chen   Advisor: Dr. Ying-Dar Lin

Department of Computer and Information Science

National Chiao Tung University

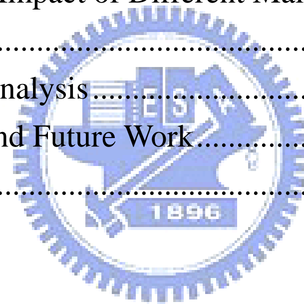
Abstract

Anti-virus systems nowadays might operate on access gateways for centralized management and early blocking viruses. When serving a group of computers, the traditional storage-based mechanism has the scalability problem due to its storage of mails under processing. This work designs a stream-based mail proxy which processes the mail segment by segment without the storage of the entire mail and interleaves the MIME parsing, decoding, decompression and virus scanning. We integrate and modify several existing open-source packages into the proxy and use the system call *select* to achieve single-process concurrency. The benchmarking reveals our proxy is seven times faster than in the storage-based mail proxy on simply forwarding, and three times faster on virus scanning, and twice faster on both virus scanning and decompression. Our proxy keeps constant memory consumption for each connection and works without disk storage while the disk usage of AMaViS is proportional to both the number of clients and the mail size.

Keywords: stream-based, segment, on-the-fly, virus, proxy, interleave, decompression

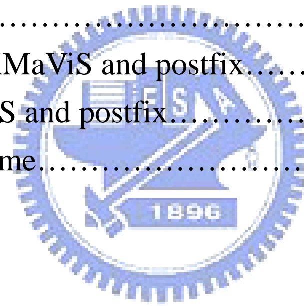
Contents

Abstract II
Contents III
List of Figures and Tables IV
Chapter 1. Introduction 1
Chapter 2. Problems and Design Issues 4
Chapter 3. System Architecture 9
3.1 System overview 9
3.2 Processing workflow 11
Chapter 4. System Implementation 14
4.1 Implementation Architecture 14
4.2 Single process concurrency 16
Chapter 5. Performance Evaluation 18
5.1 Testbed 18
5.2 Performance and the Impact of Different Mail Content 18
5.3 Buffer Requirement 21
5.4 Internal Bottleneck Analysis 25
Chapter 6. Conclusions and Future Work 27
References 29



List of Figures and Tables

Fig.1. Storage-based proxy - AMaViS……………….…………….…………………4
Fig.2. System overview………………………………………………………………10
Fig.3. Composition of a mail…………………………….…………………………...11
Fig.4. Processing mail attachments…...……………………………………………...12
Fig.5. Implementation architecture…………………………………………………..15
Fig.6. Decompression implementation………………………………………………16
Fig.7. Mail processing states…………………………………………………………17
Fig.8. Latency of sending one mail……………..……………………………………19
Fig.9. Throughput with virus scanning and decompression…...……………………..20
Fig.10. Space usage of memory and disk…………………….………………………21
Fig.11. Space usage of mail with different size………………………………………23
Fig.12. Percentage of processing time of stream-based proxy………………………25
Fig.13. Percentage of processing time of AMaViS…….……………………………26
Table 1. Compression formats…………………………………………………………7
Table 2. Programs related to AMaViS and postfix…………………………………..23
Table 3. Disk usage in AMaViS and postfix…………………………………………24
Table 4. Ratio of processing time…………………………………………………26



Chapter 1. Introduction

Conventionally, anti-virus systems run on host computers. Since most infections come from outside networks, blocking viruses on the access gateway appears to be a trend. Such a gateway-based centralized management could reduce the cost of maintaining the anti-virus system on a large number of host computers. Most of the free E-mail service provider like Yahoo![26] has the virus scanning function which needs a extra fee, and users may not take that service. To guarantee all users are protected against viruses, it needs to do anti-virus on the access gateway. Virus scanning on the gateway, however, can be *storage-based* or *stream-based*. The former receives the entire mail content before scanning, while the latter scans the part that has been received and sends it out immediately after the scanning. The storage-based scanning has bad storage scalability. For example, if 10,000 connections send 500KB files concurrently, the total storage occupied in the gateway would be 5GB. The system needs large storage and hence is more costly.

By *interleaving* the receiving, scanning and sending, the required memory buffer size for a connection can be kept *constant* rather than *proportional* to the file size. All the components in the processing flow should be also stream-based. For instance, the mail content may be MIME encoded, compressed and encrypted. Fortunately, the decoding and decompression can be stream-based, i.e. interleaved.

This work implements a stream-based mail proxy with interleaved decompressing and virus scanning. Several open-source packages are selected to be integrated: Net::SMTP::Server[1] for SMTP protocol handler and another modified version for POP3 protocol handler, ClamAV[2] for anti-virus, and Zlib[3] + Compress::Zlib[4] for decompressing. For the better performance and lower memory

usage, the system is implemented as a single-process concurrency proxy. After the implementation, we perform a series of external and internal benchmarking. This proxy is compared with AMaViS[5] in terms of throughput, latency, and the space usage in memory and disk. We intent to answer the questions: (1) How can we interleave decompression and virus scanning seamlessly, given the complex MIME mail format? (2) By how much can the stream-based proxy improve the scalability and the performance? (3) How heavy are the decompression and virus scanning compared to other components?

Related Works

Most commercial products are storage-based such as "InterScan messaging Security Suite" from TrendMicro[6], Fortigate series from Fortinet[7], and "F-pod series[8]" from FRISK Software. The open source project AMaViS is also storage-based. Until March 2005, the only product that claimed itself stream-based is the "Content Security Gateway" from CPSecure[9]. The open-source project "Anomy"[10] is a mail sanitize tool used on the F-pod antivirus product. The MIME parser in Anomy treats mails as a stream of data rather messages on disk. This concept is close to ours, but Anomy processes the attachment as an entire file. One reason that storage-based anti-virus systems still dominate the market is they can do versatile mechanisms to handle an infected file, such as quarantine. The quarantine stores the infected file so a user is able to retrieve the file. A standalone stream-based anti-virus system simply drops the infected part of the file, and the file is destroyed. To achieve functions like quarantine, the stream-based system can cooperate with another mail storage server which duplicates mails without any content analyzing.

There are several research topics about performance improvement by processing segments instead of store-and-forward. The cut-through switch[19] sends the portion of packet out before it receives the entire packet. The "segment-based proxy caching

of multimedia streams”[20] treats the whole video as variable-size segments. About the on-the-fly decompression in this paper, the implementation of “compression proxy server”[18] discussed the compression/decompression mechanism on a web proxy.

The rest of the work is organized as follows. Chapter 2 describes the requirements and design issues. The system architecture and the system workflow are presented in chapter 3. Chapter 4 gives the details of system implementation. We evaluate both stream-based and storage-based systems by external and internal benchmarking in chapter 5. Chapter 6 concludes this work.

Chapter 2. Problems and Design Issues

Overheads in a storage-based mail proxy

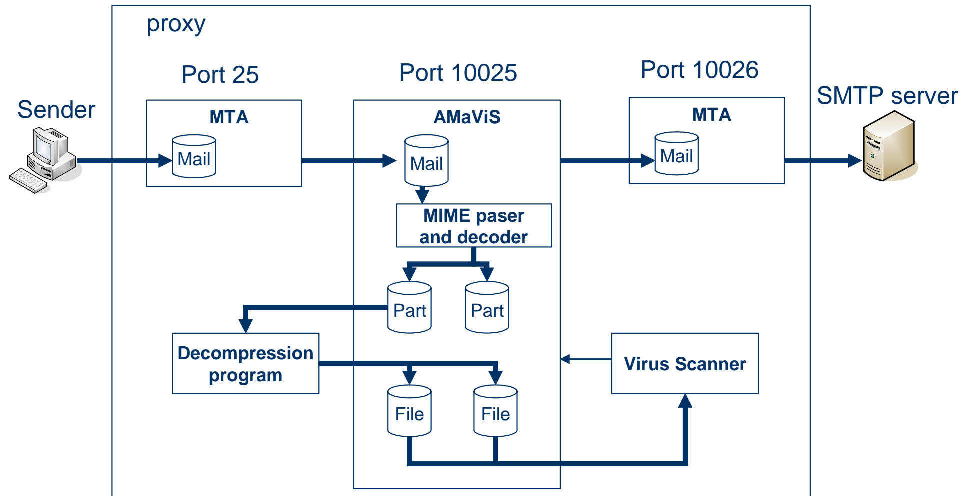


Fig.1 Storage-based proxy - AMaViS

Since AMaViS is a widely used storage-based mail proxy, we choose it to observe the mechanism and overheads in a storage-based mail proxy. Figure 1 shows the typical composition and the dataflow of AMaViS. AMaViS acts as an “interface” daemon connecting two MTA (mail transport agent) daemons. An MTA daemon receives mails from port 25. The AMaViS daemon scans the mail from the MTA. If the mail doesn’t contain virus, the AMaViS daemon transmits it to another MTA daemon which responds of sending the mail to the real target. The reasons of this complicated three daemon architecture are: (1) The historical problem, the original version of AMaViS is a script program called by MTA. It became a daemon for performance issue. (2) To protect against mail loss. AMaVis is not a full-featured SMTP server, it needs MTAs which respond of sending and receiving respectively to prevent the unpredictable things.

There exist three distinct overheads: file access, inter-process communication and process forking in AMaViS. These overheads are also examined with the

performance results in Chapter 5.

A storage-based mail proxy receives an entire file before starting to process it. It often stores the file to a disk through the file system. Any processing could be slowed down by the lengthy file system access and disk access. This overhead increases as there are other processing stages like decompression and virus scanning, all involving heavy file access. In Figure 1, the file access overhead is in all three daemons, especially in AMaViS. AMaViS receives the mail and decodes attachments into files. If the file needs to be decompressed, AMaViS calls the external program to decompress it into another file. Finally, AMaViS calls virus scanner to scan those files. Lots of file system access overhead in this processing.

Since mails need to be transferred between the three daemons, there are several inter-process communications. When AMaViS calls the external program to decompress and scan viruses, the inter-process communications also occur because the data need to be transferred between different processes.

AMaViS and most MTAs use multiple processes to achieve the concurrency. When there are many clients, per-client processes are forked in the three daemons. Lots of the memory is occupied by these processes. The *fork* system call also brings heavy overheads.

Requirements of a stream-based mail proxy

The most essential requirement of a stream-based mail proxy is that each component in the proxy should be stream-based. The processing in a mail proxy contains MIME parsing, decoding, decompressing, virus scanning, and encoding. The proxy receives a part of a mail in a memory buffer, and then processes the buffer according to its content. Some intermediate buffers may be required. For example, decompressing and decoding need extra buffers. The processing is on the buffers rather than on the entire file.

Concurrency strategy

The per-connection multi-process architecture uses too much memory. The multi-threaded architecture is more feasible. A thread can be allocated either to execute a specific function or to serve a connection. In the former strategy, each processing function in the proxy has a corresponding thread which handles many connections concurrently and also needs to synchronize with other threads. In the latter, a thread is allocated for each connection instead. Each thread handles the mail step by step, from protocol handling to virus scanning.

However, our implementation platform is on Perl. The creation of threads in Perl uses as much memory as forking processes[11] and the Perl interpreter is also duplicated, so the name of thread in Perl is "ithread" which means interpreter-level thread. Finally we choose the single-process architecture with socket I/O multiplexing to handle concurrency. Although the single-process architecture could not take advantage over the multi-processor system and is more complicated to maintain the code, it has the most economical memory usage and eliminates the context-switching overheads. There is also no thread synchronizing and inter-process communication. These could render high scalability in terms of the number of connections.

On-the-fly decompression

Storage-based systems need to store the decompressed files, which may be much larger than the original files. A denial-of-service attack could send a file that is over 100 times larger after decompression. Storage-based systems thus often bypass or block the file whose size might exceed a threshold after the decompression.

Lossless data compression methods are often the "adaptive dictionary" algorithms, such as LZ77[21], LZ78[22] and LZW[23]. A word is added to a dictionary when it appears for the first time. When the same word appears again, the encoder substitutes a short code for it. The file can be later decompressed by indexing

on the dictionary. This sequential compression/decompression mechanism makes it possible to decompress the portion of data in order. As long as the dictionary is located at the beginning of the file and the proxy receives segments in order, the stream-based decompressing by indexing should be feasible. Table 1 presents the common compression formats. The BWT[24] algorithm is block-based since it processes a block of data which is 900KB by default. The proxy need to queue the data until the entire block is received. The self-extract file contain the decompress program and the compressed data. The proxy need to identify the self-extract file and decompress it on-the-fly. Some compressed file may be encrypted, the proxy can't process the encrypted file.

Format	Program	Algorithm	File extent	Stream?
unix compress	compress	LZW	.Z	Yes
gzip	gzip	Deflate (LZ77+Huffman)	.gz .tgz	Yes
zip	Winzip	Deflate	.zip	Yes
7zip	7-zip	LZMA	.7z	Yes
rar	WinRAR	LZSS	.rar	Yes
bzip2	bzip2	BWT	.bz2	Block-based
lha	lha	LZ78+Huffman	.lha .lzh	Yes
self-extract	itself	Depends on format	.exe	Yes *

Table 1 Compression formats

The original design objectives of file compression are not for the streaming purpose. The ready-made programs and libraries all process an entire file. It makes stream-based systems not so popular in the market. To do the on-the-fly decompression, the system needs to modify low-level decompression libraries and call the low-level API directly. For example, for the files with the ".gz" extension, the *deflate* function in Zlib[] is called instead of executing the gzip[12] program. The detailed implementation is addressed in Chapter 4.

A file can be compressed more than once, i.e. recursively, and a compressed

archive may contain multiple compressed files. On-the-fly decompressing is complicate to handle the recursive compression, because it needs to parse the decompressed content continuously to check if another compressed file is there. A compressed file creates a decompressing process and a parsing process which might find another compressed file. When the archive contains multiple compressed files with recursive compression, several decompressing and parsing process at the same time and the data transfer between them is complicated. By contrast, the storage-based system can simply solve this problem by recursive decompression using external program sequentially.

Virus patterns across segment boundaries

The stream-based system scans individual buffers where segments of file content are processed, but virus patterns may be across the segment boundaries. There are two solutions to this problem. The system can keep the state of the virus scanner, i.e. which signature has its head matching the tail of last segment, through the entire scanning. This solution needs to modify the virus scanner. Another solution uses a mechanism called cushioned scanning[13]. A cushioned scan extends the buffer with sufficiently large data from the tail of the previous scan buffer on the head side. That is, data in the cushion buffer is scanned twice. The size of a cushion buffer should not be shorter than the *longest* pattern in the virus database. The same problem also occurs on decompressing, the decompression engine need to keep the decompressing status of the file throughout the entire decompressing.

Chapter 3. System Architecture

In this chapter, we present the software architecture of the stream-based mail proxy. The implementation of this architecture is described in chapter 4.

The system is designed to achieve the following goals:

Scalability: Stream-based processing is used to interleave file decompressing and virus scanning on file segments without storing the entire file. The buffer space requirement is greatly reduced. Hence, a large number of connections can be support.

Performance: A storage-based system like AMaViS often calls external commands to decompress files and scan viruses. Also, AMaViS needs to cooperate with MTAs, and so totally three daemons are on the system at the same time. The stream-based system calls the shared library to decompress and scan viruses. It is implemented in a single-process architecture. The overheads in context-switching and inter-process communication are eliminated. Also, stream-based processing eliminates the file access overheads which is especially large in AMaViS daemon described in chapter 2.

Extensibility: The system should be able to easily integrate new network protocols for extension because of separated modules. Besides the SMTP and POP3, other mail service like IMAP could be integrated in the future.

Transparency: The system monitors transparently every connection between the internal and external networks. No awareness of the system is needed.

3.1 System overview

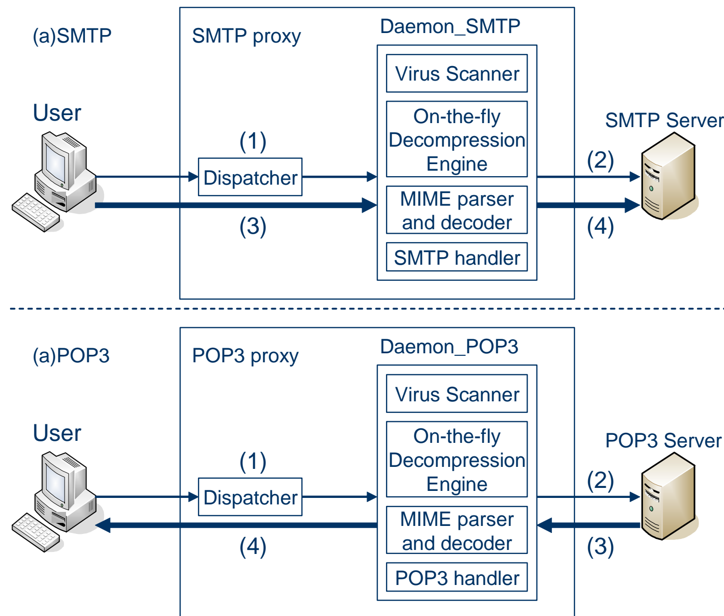


Fig.2 System overview

Figure 1 shows the overview of our system. The thin line represents the direction of protocol, while the bold line represents the direction of mail transmission. First, a dispatcher intercepts the packets from user and redirects them to the corresponding protocol handler. For example, the dispatcher redirects connections with destination port 25 to the SMTP daemon. The SMTP/POP3 handler communicates to the user and the server simultaneously. After the protocol communication, the mail is ready to be sent. The direction of mail transmission is the difference between SMTP and POP3. The data may be encoded or compressed. The attachments in a mail are encoded with MIME encoding, so the service about electronic mail like POP3 and SMTP need a MIME parser. The decoded attachment may be a compressed file, and the on-the-fly decompression engine decompresses it. After preprocessing, the system has a block or segment of partial data from the attached file. The system scans it with the virus scanner. If there is no virus, the original data read from the sender is forwarded to the receiver. If the mail contains the virus, the proxy can break the connection immediately and send a notification to user.

3.2 Processing workflow

This section presents the detailed workflow of processing one mail which is the same in SMTP and POP3. A MIME encoded mail is composed by several pairs of the MIME header and the MIME body after the mail header. The MIME header is different from the mail header. Figure 3 shows the composition of a mail. The mail body and several attachments are encoded into MIME body by several encoding methods defined in RFC 2045[25]. Common encoding methods of a MIME body are UUE, Base64, quoted-printable, etc. The MIME header contains the information of MIME body, such as the encoding method, the data type, and the filename of the attachment.

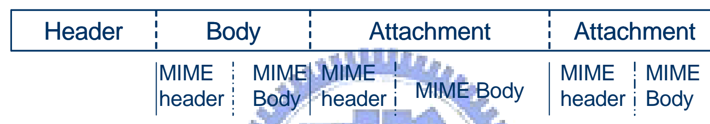


Fig.3 Composition of a MIME encoded mail

Processing the mail header

The mail header is the first part in every mail. The mail header parser reads the header from raw buffer and checks if this mail is MIME encoded. If it is MIIME encoded, the MIME parser is ready for parsing the MIME header and the MIME body.

Process mail body

The mail body is after mail header immediately. A body parser can be put here to checks the body if it is a spam, and if it contains malicious links or JAVA/VB scripts. The body parser may modify the mail body to remove these malicious things. Since we only care about the virus in attachments, the mail body is simply forwarded to the destination. There is no body parser in our implementation.

Process mail attachments

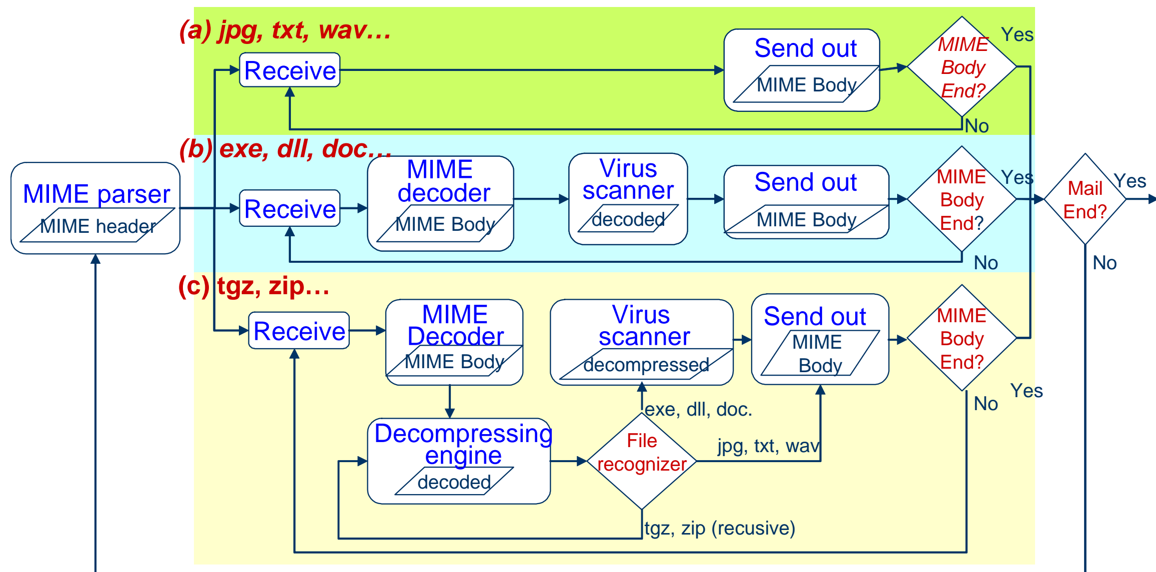


Fig.4. Process mail attachments

Attachments are mostly encoded and may be compressed. Figure 4 shows the total workflow of processing attachments. First, the MIME parser gets the file name from the MIME header. According to the file name, the proxy processes the attachment in three ways: (a) The non-malicious files, identified by the file extension, can be ignored because they could not have viruses, like “*.jpg” and “*.txt”. (b) The file type needs to be scanned for viruses such as executable files types like “*.exe” and other file types like “*.doc”. (c) If its type shows the file is compressed. The proxy needs to do decompressing before scanning. The decompressed data should also be recognized weather it may contain viruses. There is a “file recognizer” can analyze the decompressed data to decide the later process. If the decompressed data contains another compressed file, the system needs to decompress recursively. The sizes of intermediate buffers such as “decoded” and “decompressed” are not directly proportional to the size of the attachment. These buffers are created per mail. The size of “decompressed” buffer is decided by the compression ratio and the content being decompressed.

When the virus scanner finds viruses in the attachment, the proxy drop the remaining data of the attachment. The destination will receive a broken attachment.

The user on the destination is free from viruses.

Chapter 4. System Implementation

The system runs on a PC with Linux kernel version 2.6.10. It is implemented in Perl[] because of its outstanding string processing ability and various existing program libraries in Perl modules. Zlib[] is the most widely used compression/decompression library in the UNIX-like operation system. We use ClamAV[] as our virus scanner, since it is the only active open-source virus scanner at present.

4.1 Implementation Architecture

Figure 5 presents the architecture of our implementation. The bold texts are the name of modules in our system, and some names appeared in Figure 1. The names in parentheses are the existing open-source packages used in that component. All parts run within a single process in the user space. The arrows represent the relationship between components. For example, the "virus scanner interface" calls ClamAV to scan a buffer by calling *scanbuf()* in the ClamAV shared library. Except that the Zlib and ClamAV are shared libraries written in C, the other components are implemented in Perl or Perl modules.

When the kernel receives the packets, *netfilter* redirects the packets with destination port 25 (used by SMTP) or port 110 (used by POP3) to the port our proxy server is listening on. The proxy server accepts the connection and identifies a socket handler. After the SMTP handler communicates with the socket handler from SMTP sender, it connects to the SMTP target to get another socket handler. With both the source and target socket handlers, a mail processor is created. The mail processor is

written as a module which can be created as an object in run time.

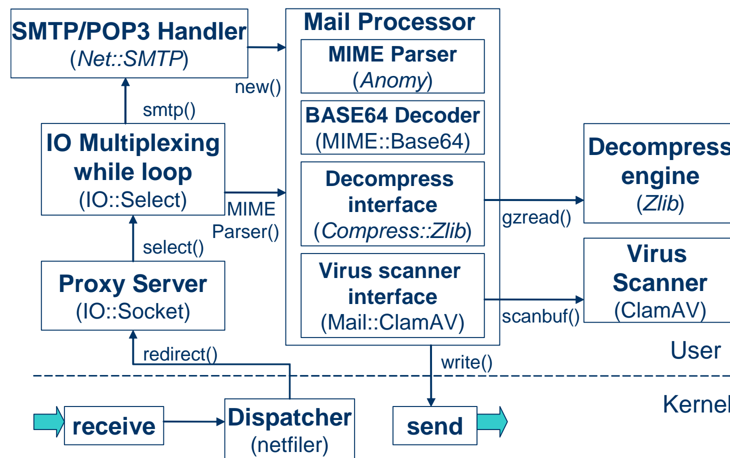


Fig.5 Implementation architecture

The mail processor handles the entire mail, including parsing MIME, reading the buffer from the source socket, scanning the buffer and writing the buffer to the target socket. The MIME parser in the mail processor is an open-source package "Anomy" which is a mail sanitizer. Because every connection creates a mail processor object, it becomes the main overhead in the memory when there are a large number of connections. The mail processor is independent of any protocol. To monitor the POP3 service, we can simply use the POP3 handler to cooperate with the mail processor. The detailed workflow of the mail processor is presented in section 3.2.

The italic type of the open-source package in Figure 6 means the codes of that package are modified for our purpose, including Net::SMTP::Server, Compress::Zlib and Zlib. Because of IO multiplexing, we modify Net::SMTP::Server to process one line a time whenever a socket is selected. Compress::Zlib is a Perl module and an interface to call the Zlib shared library in Perl. Zlib fails if it reads the end of data stream which does not equal to the end of file. We remove this limit in Zlib to make partial decompression possible. Other packages without modification can be upgraded to newer versions if the arguments of the functions used in the package remain their original definition.

Fig.6 Decompression implementation

Figure 6 shows the detailed implementation of decompression. The system supports the files compressed by the Zlib at first in our implementation. *Gzopen* and *gzread* are functions in the Zlib shared library. Because Zlib is designed to decompress an entire file, it opens the file handler by *gzopen* function before any decompressing using *gzread*. The system treats the handler "handler_out" as the file opened by Zlib, and input the decoded data into the handler "handler_in." The handler "handler_in" connects with the handler "handler_out" by the inter-process communication mechanism called Pipe. The handler "handler_in" needs to be set as "non-blcoking I/O", and therefore we are able to input the decoded data to the handler "handler_in" and decompress data from the handler "handler_out" in turn. The combination of handler_in, Pipe, and "handler_out" can be seen as a queue which is capable of reading and writing at any time. This mechanism is applicable to all to every compression libraries originally designed to handle an entire file.

4.2 Single process concurrency

We implement the server as a single process and use *select()* to achieve concurrency, because both of the multi-process and multi-thread mechanisms in Perl consume lots of memory resource. The detailed reason is described in chapter 2.

Because only one process handles all clients in turn, we need to keep the state of every client. Every time when I/O multiplexing selects a client to handle, the system calls the corresponding function according to the *state* of clients. The functions in Figure 6 are smtp() and MIME parser(). We can derive Figure 7 to show all states of a client from all the mail processing situations in chapter 3. Except that the SMTP and

“quit or next” states are related to the SMTP protocol, other states are kinds of the MIME parsing states. ”Bypass”, “scan” and “decompress” handle the attachment in three ways described in Section 3.2.

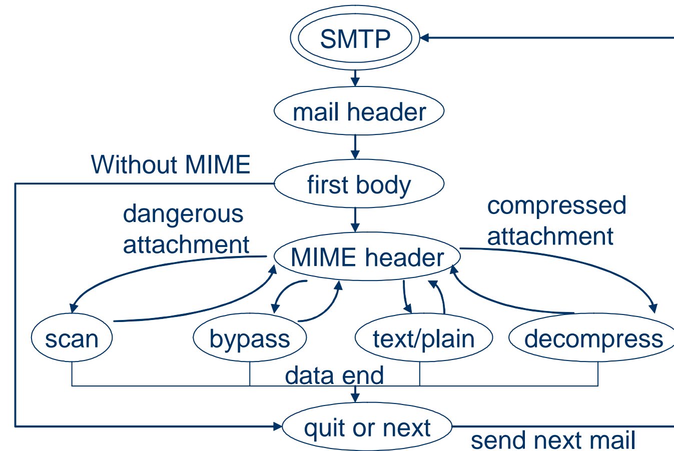


Fig.7 Mail processing states

To achieve short response time, the processing time in each state should be short. The SMTP protocol handler handles one protocol message at a time in the SMTP state. The system reads only 8KB data each time when handling the three types of attachment. In AMaViS, however, the MTA receives all mails and store them to the disk first. Then AMaViS processes mails sequentially. If there is a large file in front of many small mails, small mails need to wait until the large one has been finished. The average processing latency in the storage-based proxy may be long because the large mail blocks the small mail. The stream-based proxy often has shorter latency and servers clients fairly.

Chapter 5. Performance Evaluation

5.1 Testbed

We compare our stream-based mail proxy with AMaViS that is a storage-based mail proxy. We install these two proxies on a PC with 1GHz PentiumIII CPU, 512MB SDRAM, 20GB hard disk and 100Mbps Ethernet network. The operating system is Linux with kernel version 2.6.10. We use Perl 5.8.5 to run both proxies which are both implemented in Perl. Both proxies use ClamAV 0.83 as the virus scanning engine. Because AMaViS is an interface to cooperate with two MTAs, we use Postfix since it fully supports AMaViS.

For fairness, we configure AMaViS in the following way: (1) disable the anti-spam function since our stream-based proxy does not check the spam mail, (2) run ClamAV in the daemon mode which is faster than the command line mode, (3) disable the cache mechanism since AMaViS bypasses the same mail processed before within a configurable time.

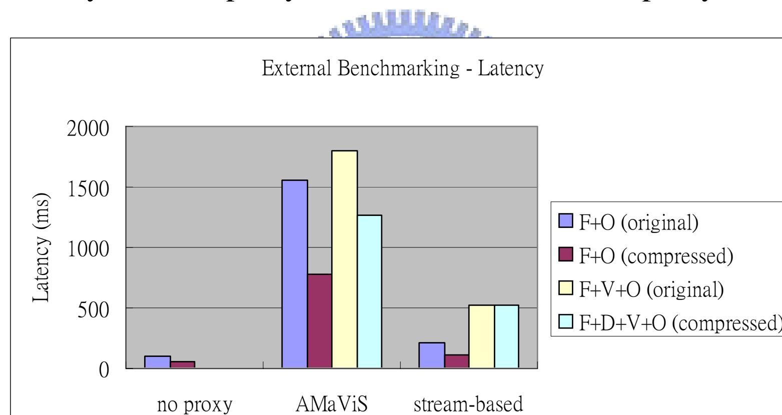
We prepare two types of mails as the mail traffic in our benchmarking to test different processing mechanisms. The first is the mail with 1MB executable attachment and will not to be scanned for virus or decompressed. The proxy simply forwards this mail. The second is the mail attaching the compressed file from the previous 1MB executable file. The compression ratio is 37%. The size of the first mail is 2.71 times of the second mail. Because both proxy scans the decompressed attachment, these two mail have the same content to be scanned.

5.2 Performance and the Impact of Different Mail Content

To understand the difference in performance between the stream-based mail proxy and the storage-based mail proxy, we measure latency and throughput. Three types of mail traffic are used.

Since AMaViS receives the mail and stores it to the hard disk first before processing, the mail sender finishes sending before the start of receiving on the target. We need to log the end of receiving on the target rather than the end of sending on the mail sender. The mail sender and the target receiver are run on the same computer, so we are sure that the times logged on sender and receiver use the same time clock.

Latency is the time from the start of sending one mail to the end of receiving on the target MTA. When the proxy is used, the mail is held by the proxy for a while. We observe the latency with our proxy, AMaViS and without the proxy environment.



F: forwarding O: other mail processing V: virus scanning D: decompression

Fig.8 Latency of sending one mail

Figure 9 shows the results of latency. The forwarding time are tested on both mails in three proxy environments. The latency is 102 ms without extra processing or the proxy. When the proxy simply forwards the mails, our proxy takes 213 ms and 105 ms, while AMaViS takes 1553 ms and 780 ms. Compared with virus scanning and decompressing, the latency of the our proxy mail in our proxy is 518 ms and 527 shorter than 1802 ms and 1267 of AMaViS. The result also means AMaViS is more sensitive of mail size than our proxy. Significantly, our proxy has short latency in all

types of mails we tested.

Throughput is defined as the total mail size divided by the elapsed time. A large number of identical mails are sent through the proxy and the total elapsed time is measured. The size of a mail is different from that of the file attached since Base64 encoding expands the size of the file being attached to 1.33 times. We use the size of the mail to calculate the throughput. To achieve the maximum throughput, we use more than twenty clients on the sender sending to our proxy concurrently. Since AMaViS receives all mails and then processes sequentially regardless of the number of clients, a large number of clients do not have larger throughput on AMaViS.

The throughput of our proxy when the proxy simply forwards maisl is 65.2 Mbps which is very close to the throughput of 69.93 Mbps without any proxy. AMaViS gets the throughput of 9.51 Mbps even when it disables both anti-virus and anti-spam functions. We can conclude that the storage-based architecture itself is a bottleneck.

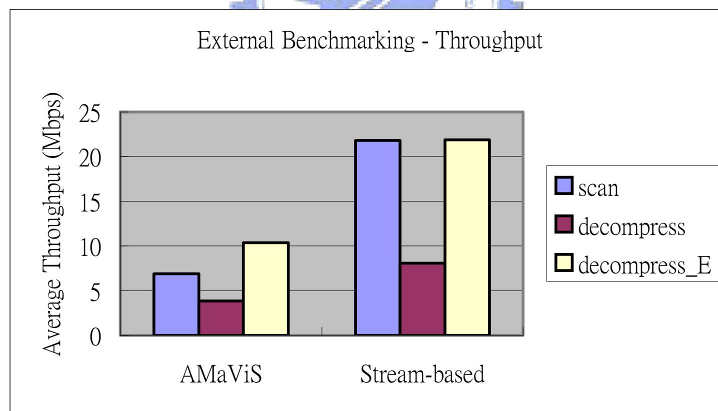


Fig.9 Throughput with virus scanning and decompression

Figure 10 shows the throughput with virus scanning and decompression. With virus scanning but without decompression, our proxy has 21.79 Mbps. Dropping from 65.2Mbps in simple forwarding implies virus scanning is the bottleneck. AMaViS gets 6.9 Mbps with virus scanning, slightly dropped from 9.51 Mbps in simple forwarding. The mail with a compressed attachment has two throughput values. The higher one is the “effective throughput”, denoted with “_E”, to represent the throughput in scanning

the decompressed file, calculated with the decompressed attachment size instead of the mail size. Because the file size to scan for viruses is the decompressed attachment size, the effective throughput represents the real throughput of virus scanning.

From external benchmarking, we conclude the following points. (1)Our proxy has a much better performance the AMaViS. (2) The storage-based architecture itself is a bottleneck. (3) Virus scanning takes more time than decompression.

5.3 Buffer Requirement

We evaluate the total buffer size by monitoring the disk and memory consumption of two proxies while there are variable clients. Each client sends one mail attaching a 300K file compressed from a 1MB file. Figure 11 shows the result.

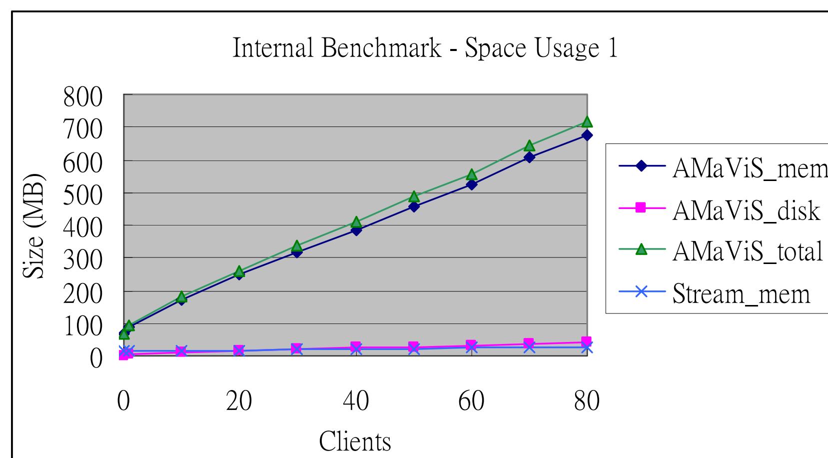


Fig.10 Space usage of memory and disk

"AMaViS_mem" and "AMaViS_disk" mean the memory usage and disk usage, respectively, on AMaViS and postfix. Because AMaViS needs to cooperate with postfix, we count them both. "AMaViS_total" is the sum of "AMaViS_mem" and "AMaViS_disk". "Stream_mem" means the memory usage of our proxy. Since we don't use any temporary file, there is no disk usage of our proxy. We can figure out the storage-based proxy uses much more space on both memory and disk than our stream-based proxy.

In the system, there are two kinds of buffer requirement: the runtime process space and the mail-storage space. The runtime process space is required when the forking of process and absolutely is in the memory. In a per-connection multi-process architecture, the runtime process space is directly proportional to the number of clients. The mail-storage space is required when the proxy processes the mail, and it might be in the disk or the memory. The mail-storage space is often directly proportional to both the mail size and the client number in the storage-based system.

The memory usage glows enormously in the combination of AMaViS and postfix because of the complicated communication between the three daemons described in Chapter 2. The memory usage in AMaVis and postfix is the runtime process space. Both of postfix and AMaViS have multiple processes, and they fork the corresponding number of processes to handle clients. First postfix daemon receives mails from the clients and sends mails to AMaViS, and second postfix daemon receives mails from AMaViS and sends mails to real target. The number of AMaViS is configured before running the proxy and is a fixed number in system run time. The number of postfix child processes is the sum of the number of clients and the number of AMaViS. An SMTP sending program is used to send mail to AMaViS processes by the first postfix daemon and the real target by the second postfix daemon. Table 1 lists all programs related to AMaViS and postfix, including the size and the number of the processes. We use X to represent the number of clients and Y to represent the number of AMaViS child processes. The memory usage of AMaViS and postfix is

$$(4491+2859) * (X+Y) + 4259*2Y + 20430*Y + 19000 + 2759 + 7463$$

The memory usage grows about 7350KB per client in AMaViS and postfix, it is the sum of the "smtpd" program memory usage and the "cleanup" program memory usage. The number of "Cleanup" processes is corresponding to the number of "smtpd" processes which increases as the increasing of the number of clients.

Program	Description	Size	Number
smtpd	Postfix SMTP server child process	4491	X+Y
cleanup	Process the queue received by smtpd	2859	X+Y
smtp	Postfix SMTP sender	4259	2Y
AMaViS child	AMaViS child process	20430	Y
AMaViS master	AMaViS listening on port 10025	19000	1
Postfix master	Postfix listening on port 25 and 10026	2759	1
Clamd	ClamAV daemon	7463	1
X : The number of clients Y: The number of AMaViS child processes(fixed during run-time)			

Table 2 Programs related to AMaViS and postfix

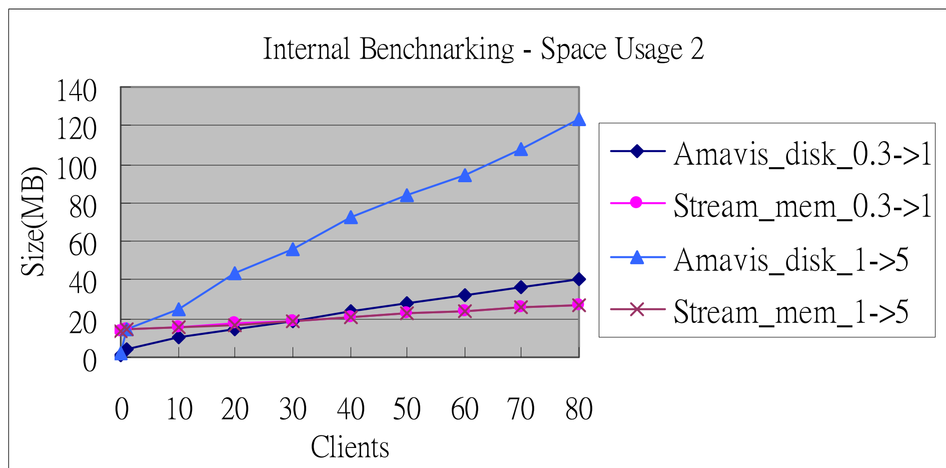


Fig.11 Space usage of mails with different sizes

Figure 12 removes “AMaViS_mem” and “AMaViS_total” in Figure 12, and adds another result tested by mails attaching a 1MB file compressed from a 5MB file. The disk usage “AMaViS_disk” is used to store mails being processing and is the mail-storage space. The disk usage in storage-based proxy is directly proportaional to the mail size. We can see the difference in AMaViS when using different mail size. Our proxy remains the same memory usage no matter how large the mail size is because of the streaming operation with interleaved decompression and virus scanning. Our proxy has 13.7MB runtime process space in memory when there is no client. The runtime process space in our proxy does not increase as the increasing of

the number of clients, because of the single-process architecture. The memory usage in our proxy only increases 176KB per client and the increasing is the mail-storage space. The 176KB is the mail processor described in section 4.1, it is composed of buffers and variables to record mail states.

	Description	Size	Number
Postfix	Store all mails on disk	452 (mail size)	X
AMaViS	Save the mail being processed	452	Y
	Decompress the file	1032 (decompressed)	Y
	Copy the files from the archive	<1032	Y
X : The number of clients Y: The number of AMaViS child processes(fixed during run-time)			

Table 3 Disk usage in AMaViS and postfix

Table 2 lists the detailed disk usage when using the mail attaching a 300K file decompressed from a 1MB file. After postfix receives all the mails, AMaViS saves one copy of each mail in its repository. Then AMaViS calls the external program to decompress the file from the original archives, and copy the selected type of files from archives to scan for viruses. We can use another equation to calculate the disk usage in AMaViS and postfix as

$$(\text{Mail Size}) * X + (\text{Mail Size} + 2*(\text{Decompressed Size})) * Y$$

We monitor the memory space usage and disk usage on both proxies to determine the requirement of the runtime process space and the mail-storage space. The runtime process space of AMaViS grows enormously as the increasing of the number of clients, because of the complicated architecture. The mail-storage space of AMaViS is in the disk and is directly proportional to both the number of clients and the mail size. The runtime process space of our proxy is about 13.7MB regardless of the number of clients, because of the single-process architecture. The mail-storage space of our proxy is in memory and is about 176KB per client regardless of the mail size.

5.4 Internal Bottleneck Analysis

To verify the bottleneck more clearly, we use the Perl module Devel::Profile[16] to record the processing time of every function in our proxy. We tested several mails attaching 1MB executable file and attaching decompressed file of that 1MB executable file. From Figure 12 we can clearly figure out the bottleneck is virus scanning which takes above 60% of the execution time. Although the decompression is not the main bottleneck in our proxy, given two identical size mails, one is compressed and the other is not, the compressed one takes more time on scanning for viruses because the proxy scans the file size after the decompression. The main bottleneck in virus scanning is matching the virus patterns. If we can improve the string matching algorithm in virus scanner, the throughput of our proxy can be improved. Figure 13 shows the internal processing time of AMaViS. The file system access overheads are in virus scanning, decompression, receiving, sending, and IPC. If we set the ratio of decompression to 1, the ratio of processing time is presented in Table 4.

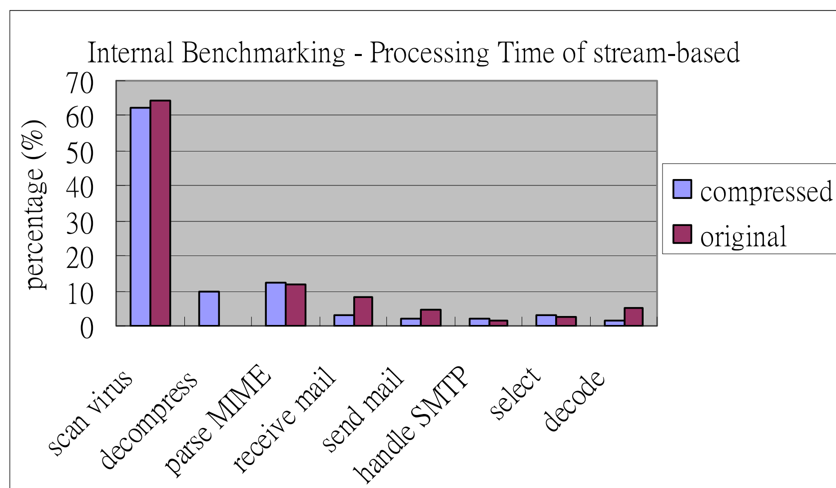


Fig.12 percentage of processing time in stream-based mail proxy

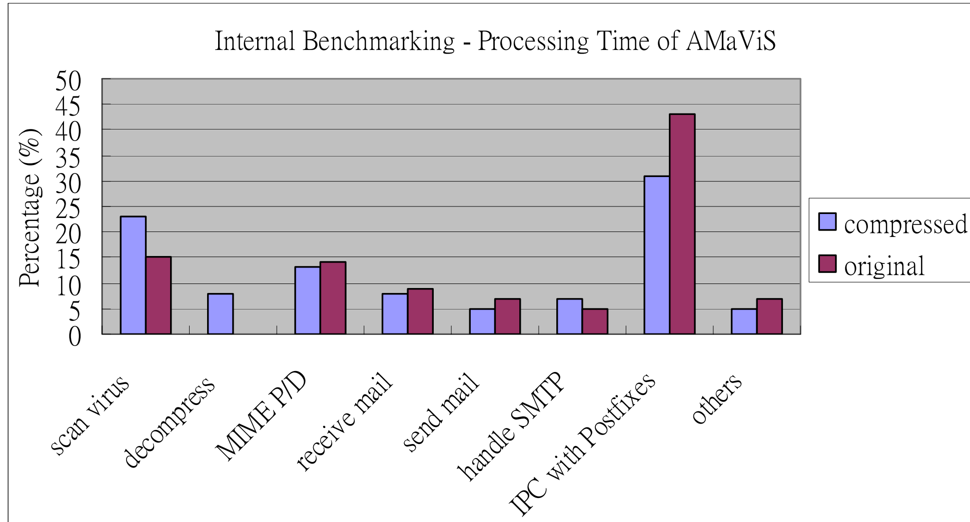
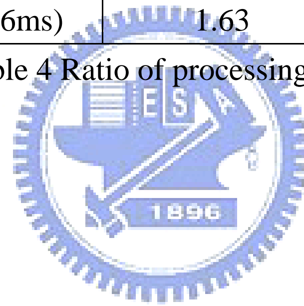


Fig.13 percentage of processing time in AMaViS

	scan	decompress	Handle MIME	receive	send	IPC
Stream-based	6.3	1 (55ms)	1.44	0.32	0.22	
AMaViS	2.88	1 (96ms)	1.63	1	0.63	4.25

Table 4 Ratio of processing time



Chapter 6. Conclusions and Future Work

In this work, we design and implement a stream-based mail proxy with interleaved decompression and virus scanning to avoid storing an entire mail. Without storing the entire mail, we eliminate the file system access and save the buffer usage. Several benchmarking experiments compare the storage-based proxy with our stream-based proxy in performance and space usage. An internal profiling analyzes the bottleneck of our system.

The external benchmarking shows our proxy has shorter latency and higher throughput in both mail with the original file attached and mail with compressed file attached. When the proxy just forwards the mail to the target, the decreased percentage of the throughput is 6.7% from 69.93 Mbps to 65.2 Mbps in our proxy while it is 86.4% form 69.93 Mbps to 9.51 Mbps in AMaViS. Our proxy has 21.79 Mbps more than 6.9 Mbps in AMaViS when scanning mail for the virus, and has 8.05 Mbps more than 3.82 when scanning and decompression. In the space usage, our proxy grows 176KB per client in memory while the storage-based proxy grows 7350KB, and our proxy does not use any temporary file on disks while the disk usage of storage-based proxy is directly proportional to both the number of client and the mail size. Consequently, our proxy is better on both speed and space usage. The file size to be scanned for viruses dominates the processing time in both proxies, and virus scanning is the main bottleneck in our system.

This system is feasible for the embedded system environment without a hard disk and is more scalable than the traditional storage-based proxy. Designing a better algorithm or a hardware accelerator of string matching in the virus scanner can speed up the system. Anti-spam is another useful function in the mail proxy, and we can do

it when processing mail body. Another way to improve system is that implement the system in C instead of Perl. C is faster but has a worse string processing ability.

References

[1] Perl module: Net::SMTP::Server, http://search.cpan.org/~macgyver/SMTP-Server-1.1/Server.pm .

[2] Clam AntiVirus, http://www.clamav.net/ .

[3] Zlib, http://www.gzip.org/zlib/ .

[4] Perl module: Comperss::Zlib, http://search.cpan.org/~pmqs/Compress-Zlib-1.34/Zlib.pm .

[5] AMaVis – A Mail Virus Scanner, http://www.amavis.org/ .

[6] Trend Micro, http://www.trendmicro.com .

[7] FORTINET, http://www.fortinet.com/ .

[8] F-pod Antivirus, http://www.f-prot.com/ .

[9] CP Secure, http://www.cpsecure.com/ .

[10] The Anomy mail tools, http://mailtools.anomy.net/ .

[11] Things you need to know before programming Perl ithreads, http://qs321.pair.com/~monkads/index.pl?replies=1&node_id=288022&displaytype=print .

[12] gzip, http://www.gzip.org/ .

[13] Yevgeniy Miretskiy, Abhijith Das, Charles P. Wright, and Erez Zadok, "*Avfs: An On-Access Anti-Virus File System*", The 13th USENIX Security Symposium, 2004.

[14] RFC 3548 - The Base16, Base32, and Base64 Data Encodings, http://www.faqs.org/rfcs/rfc3548.html .

[15] The C10K problem, http://www.kegel.com/c10k.html .

[16] Perl module: Devel::Profile, http://search.cpan.org/~jaw/Devel-Profile-1.04/Profile.pm .

[17] Ying-Dar Lin, Chih-Wei Jan, Po-Ching Lin, and Yuan-Cheng Lai, "*An Integrated Proxy Architecture for Anti-Virus, Anti-spam, Intrusion Detection, and Content Filter*"

[18] Chi-Hung Chi, Jing Deng, Yan-Hong Lim, *"Compression Proxy Server: Design and Implementation"*, USENIX Internet Technologies & Systems, 1999

[19] P Kermani, L Kleinrock, "*Virtual Cut-Through: A New Computer Communication Switching Technique*" - Computer Networks 3, 1979

[20] K. Wu, P. S. Yu and J. L. Wolf, "*Segment-based Proxy Caching of Multimedia Streams*", WWW'2001, pp. 36-44.

[21] Ziv, J., Lempel, A., "*A universal algorithm for sequential data compression*," IEEE Transactions on Information Theory, IT-23:337-343, 1977.

[22] Ziv, J., Lempel, A., "*Compression of individual sequences via variable-rate coding,*" IEEE Transactions on Information Theory, IT-24, 5, 1978.

[23] Welch, T.A., "*A technique for high-performance data compression*," Computer 17, 6 (June 1984), 8-19.

[24] M. Burrows and D. J. Wheeler, "*A block-sorting lossless data compression algorithm,*" Digital SRC Report 124, 1994.

[25] RFC 2045 - Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies, http://www.faqs.org/rfcs/rfc2045.html .

[26] Yahoo! Mail, http://mail.yahoo.com/ .