Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 81 (2008) 1517–1524
The Journal of Systems and Software
www.elsevier.com/locate/jss

Designing and evaluating interleaving decompressing and virus scanning in a stream-based mail proxy

Ying-Dar Lin [a], Szu-Hao Chen [a], Po-Ching Lin [a,*], Yuan-Cheng Lai [b]

[a] Department of Computer Science, National Chiao Tung, University, No. 1001, Da-Hsueh Road, Hsinchu 300, Taiwan
[b] Department of Information Management, National Taiwan University of Science and Technology, No. 43, Section 4, Keelung Road, Taipei 106, Taiwan

Received 14 January 2006; received in revised form 9 September 2007; accepted 5 October 2007
Available online 17 October 2007

Abstract

A storage-based anti-virus access gateway is not scalable because it stores the entire mail under processing. This work designs and evaluates a stream-based mail proxy constructed from several open-source packages. This proxy processes mail in segments, and interleaves MIME parsing, decoding, decompressing and virus scanning. It is seven times faster than the storage-based one on forwarding, three times faster on virus scanning, and twice as faster on decompressing plus virus scanning. This proxy can keep nearly constant memory usage and work without disks, while the storage-based one requires memory and disk space proportional to the number of clients and the mail size.

Keywords: Mail proxy; Virus scanning; Decompression; Stream-based; Interleaving

1. Introduction

Anti-virus programs conventionally run on host computers or servers, but not all users install one and update the signatures in time. To protect all inside users from outside viruses, blocking viruses on the access gateway appears to be a promising approach for centralized management that can reduce the maintenance cost. Virus scanning on the gateway can be *storage-based* or *stream-based*. The former receives and stores the entire content before virus scanning, while the latter keeps only the part under processing and sends it out immediately after processing. The former is not scalable because it needs large disk space and the latency is long. The demand for large disk space makes impossible a diskless design that can be found in many SOHO (Small Office, Home Office) devices. Moreover, disk accessing is much slower than memory accessing, so the performance of a storage-based system is degraded. Although using a RAM disk can prevent disk accessing from degrading the performance, the scalability problem still exists because the demanded space may exceed the size of physical memory.

Most commercial products are storage-based, such as *InterScan Messaging Security Suite* from TrendMicro (http://www.trendmicro.com) and the *FortiGate* series from Fortinet (http://www.fortinet.com). Few products to date as we know are claimed to be stream-based, e.g., the *CSG* series from CPSecure (http://www.cpsecure.com). Storage-based anti-virus systems still dominate the market perhaps because they can handle an infected file in versatile ways, such as quarantine that stores an infected file so that the user can retrieve it later. A stream-based system simply drops the infected part of a file, so the file cannot be used any longer. Besides, adding a new function to a storage-based system is easier, e.g., supporting a new compression format. Despite a few minor drawbacks, a stream-based system is tantalizing for its high scalability.

Despite existing stream-based products, little information about their mechanisms is revealed. For example, the

* Corresponding author. Tel.: +886 3 5731899; fax: +886 3 5721490.
E-mail addresses: ydlin@cis.nctu.edu.tw (Y.-D. Lin), shchen@cis.nctu.edu.tw (S.-H. Chen), pclin@cis.nctu.edu.tw (P.-C. Lin), laiyc@cs.ntust.edu.tw (Y.-C. Lai).

0164-1212/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.10.003

CSG series of CPSecure overlaps the receiving, scanning and outputting stages with pipelining, and assigns a thread to handle the packet output (CPSecure, 2006). The *ZyWall UTM* products of ZyXEL can scan individual packets without packet reassembly (ZyXEL and Kaspersky, 2006). However, many details of them are not clear, such as how a compressed file is handled and scanned, and whether a dedicated thread handles packet output from all connections or an individual thread for each connection.

Stream-based processing to increase performance has existed for decades. An early work is the cut-through switch (Kermani and Kleinrock, 1979) that can send out a portion of a packet before it receives the entire packet. The segment-based proxy cache mechanism (Wu et al., 2001) can cache multimedia streams in variable-sized segments and reduce the latency of playback. The compression proxy server in (Chi et al., 1999) can select a compression mechanism for distinct class of Web objects, such as the *gif* file and data stream. Nonetheless, little research literature to date, if not none, addresses the issues of stream-based virus scanning by interleaving each step in the processing.

This work designs and evaluates a workable software architecture of a stream-based mail proxy that can interleave receiving, MIME parsing, decoding, decompressing, virus scanning and sending of each segment during mail processing. Although mail processing is not the sole application of stream-based virus scanning, it can typify the flow of stream-based operation. Unlike known commercial products that are black boxes, this design is *completely* based on open-source packages, such as *Net::SMTP::Server* (http://www.cpan.org) for SMTP protocol handler and a modified version of POP3 protocol handler, *ClamAV* (http://www.clamav.net) for virus scanning and *Zlib* (http://www.zlib.net) plus *Compress::Zlib* (http://www.cpan.org) for decompressing. The system is implemented as a single-process concurrent proxy. The short interleaving processing of each mail segment allows instant switching between clients and makes possible single-process concurrency.

This paper discusses the design issues of the seamless interleaving processing, illustrates the processing flow, and evaluates the scalability and performance of the new design by performing a series of external and internal benchmarks. The proposed proxy is compared with a popular storage-based open-source mail virus scanner, *AMaViS* (http://www.amavis.org) in terms of throughput, latency and space usage in memory and disk. The rest of the paper is organized as follows. Section 2 discusses the design issues. The system architecture and its implementation are presented in Section 3. The evaluation of both the stream-based and the storage-based systems by external and internal benchmarks is presented in Section 4. Section 5 concludes this work.

2. Design issues

It is essential that each step is also stream-based during the processing in a stream-based mail proxy. The proxy should store part of a mail in a memory buffer and complete the entire processing in the buffer. Some arising design issues are discussed below.

Concurrency strategy: A multi-process architecture is not scalable to handle multiple connections. Although multi-threading is a lightweight alternative, we choose a single-process architecture with socket I/O multiplexing to handle concurrency for two reasons. First, the Perl interpreter is duplicated for each new thread in the current Perl distribution (Liz, 2003), and the duplication is as heavy as forking a new process. Hence multi-threading does not benefit our current implementation in Perl. Second, although the single-process architecture could not take advantage of a multi-processor system and could be more complicated to maintain the code, it consumes the least memory space and avoids the overheads of context switching and thread synchronization. This architecture is feasible because of the short processing time of mail segments instead of an entire mail. The I/O operation is also made non-blocking. The design is therefore highly scalable for numerous connections.

On-the-fly decompression: A storage-based system needs to store a decompressed file that could be much larger than the compressed one. A denial-of-service attack could send a file that will be expanded over 100 times larger after the decompression. A storage-based system thus often bypasses or blocks a file larger than a specified size.

Stream-based decompression is feasible because lossless data compression methods are often *adaptive dictionary* algorithms, such as LZ77 (Ziv and Lempel, 1977) and LZW (Welch, 1984). Each word is added into a dictionary in its first appearance. When the same word reappears, the encoder substitutes a short code for it. The file can be later decompressed by indexing on this dictionary. This sequential compression/decompression mechanism makes possible decompressing portions of data in order. As long as the dictionary is located in the beginning of the file, and the proxy receives ordered segments, stream-based decompression should be feasible. Table 1 summarizes the feasibility for common compression formats. Among them, the BWT algorithm (Burrows and Wheeler, 1994) has to process data in blocks of 900 kB by default. The proxy needs to queue the data until the entire block is received. A self-extracting file embeds the decompression code in the compressed file. The proxy needs to identify the code and then decompresses the file with it.

Two cases need special treatment. A file can be compressed more than once, i.e., recursively. A compressed archive may contain multiple compressed files, e.g., the files with the extension ".tar.gz" on Unix. On-the-fly decompression needs to parse the decompressed content recursively to check if another compressed file exists to handle such recursive compression. In comparison, a storage-based system can simply solve this problem by recursively invoking an external decompression program. A common exception that cannot be handled by both types of proxies is an encrypted file. The proxy cannot deal with an encrypted file without the keys.

Table 1
Feasibility of stream-based decompression for common compression formats

Format	Algorithm	File extension	Stream-based?
UNIX compress	LZW	.Z	Yes
Gzip	Deflate(LZ77 + Huffman)	.gz or.tgz	Yes
Zip	Deflate	.zip	Yes
7zip	LZMA	.7z	Yes
Rar	LZSS	.rar	Yes
Bzip2	BWT	.bz2	Yes (in blocks)
Lha	LZ78 + Huffman	.lha or.lzh	Yes
Self-extracting	Depends on the format	.exe	Yes

We modify the low-level decompression libraries in this design for on-the-fly decompression. The low-level APIs are called directly. For example, for the files with the ".gz" extension, the modified deflation function in *Zlib* is called directly without executing the *gzip* (http://www.gzip.org) program. The implementation is detailed in Section 3.

Virus patterns across segment boundaries: A stream-based system sequentially scans individual buffers in which segments of file content are processed, but virus signatures may exist across the segment boundaries. Two solutions can address this problem. The system can keep the state of the virus scanner, i.e., which signature has its prefix matched with the tail of the last segment and the matched portion. Another solution is a mechanism called *cushioned scanning* (Miretskiy et al., 2004). A cushioned scan extends the buffer and copies sufficiently large data from the tail of the previous buffer to 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. A similar problem also occurs on decompression. The decompression engine needs to keep the decompressing status throughout the decompressing process.

3. System architecture and implementation

This system is designed to meet the following goals. (1) Scalability: The demanded buffer space can be greatly reduced to support a large number of connections by interleaving file decompressing and virus scanning on segments without storing the entire file. (2) Performance: The overheads in context switching and inter-process communication are eliminated due to the single-process architecture. Besides, the stream-based processing all operates in memory without disk accessing. (3) Extensibility: The system should be extensible for new network protocols by modularization. Besides the SMTP and POP3, other mail services, say IMAP, could be integrated as a new module in the future. (4) Transparency: The system should transparently monitor every connection between the internal and external networks.

3.1. System overview and processing flow

Fig. 1 illustrates the overview of our system. First, a dispatcher intercepts the packets from the user and redirects them to the right protocol handler. For example, the dispatcher redirects packets with destination port 25 to the SMTP daemon. The SMTP or POP3 handler communicates with the user and the server simultaneously. The two handlers differ in the direction of mail transmission. The MIME parser decodes and analyzes the MIME encoding in the mail attachments. The on-the-fly decompression engine can decompress an attached file if it is compressed. After the preprocessing, the system has a segment of partial data from the attached file. If no virus is found, the mail from the sender is forwarded to the receiver; otherwise, the proxy will overwrite the infected part and its remaining data of the attached file. Note that if the proxy breaks the connection immediately, the server considers this situation a failure and may keep trying to send the mail with viruses.

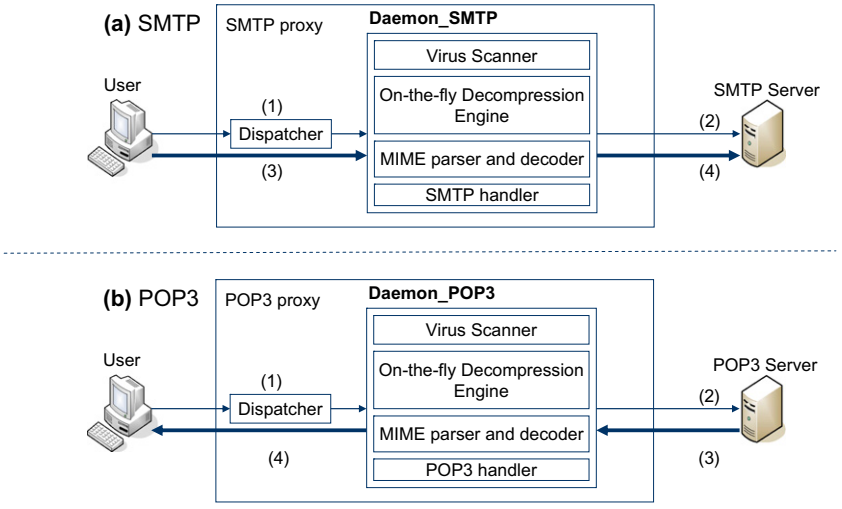

Fig. 1. System overview of the stream-based architecture. The thin line represents the direction of protocol commands, and the bold line represents the direction of mail transmission.

Hence we choose to transparently overwrite the message herein.

A MIME encoded mail consists of the mail header followed by the mail body and the attachment(s), if not none. Each mail body and each attachment are composed of a MIME header and a MIME body encoded by the methods defined in RFC 2045 (Freed and Borenstein, 1996), including UUE, BASE64, quoted-printable, etc. The MIME header contains the information of MIME body, such as the encoding method, the data type and the file name of the attachment. The processing flow of each component is described below.

Processing the mail header: The mail header is the first part in a mail. A header parser checks if the mail is MIME encoded. If it is, the MIME parser is ready for parsing the MIME encoding.

Processing the mail body: The mail body follows the header. A body parser and a spam filter can be added to check if the body is a spam. Since we only intend to scan viruses, no such parser or filter is in this implementation. The mail body is simply forwarded to the destination.

Processing the mail attachments: Fig. 2 shows the flow of processing attachments in three possible ways according to the file name from the MIME header. (a) The non-malicious files identified by the file extension, such as "jpg" and "txt", are ignored since they could not contain viruses. (b) The files identified as executable files, MS-Word documents and so on should be scanned for viruses. (c) If a file is compressed, the proxy decompresses the file before scanning. The file recognizer can analyze the decompressed data to check if more compressed files are embedded. If so, the proxy will decompress the data recursively. The sizes of intermediate buffers for decoding and decompressing are not proportional to the attachment size, but are decided by the compression rate and the content being decompressed. The decompressed data should be also checked for viruses if necessary. When the virus scanner finds a virus in the attachment, the proxy overwrites the remaining data of the attachment with null characters (including the segment with the virus), and thus the destination is unable to interpret the broken attachment. Because the attached file cannot be successfully opened, it will bring no harm to the destination. The proxy also sends the user a message to notify the broken attachment.

3.2. System implementation

The system runs on a Linux system. It is implemented in Perl due to Perl's outstanding ability of string processing and rich libraries in Perl modules. Fig. 3 presents the software components in the system. Except *Zlib* and *ClamAV* that are shared libraries written in C, the other components are implemented as Perl modules. The arrows represent the relationship between components. For example, the virus scanner interface calls *scanbuf( )* in the *ClamAV* shared library to scan a buffer. All the components run within a single process in the user space.

When the kernel receives the packets, *netfilter* redirects the packets with destination port 25 (for SMTP) or port 110 (for POP3) to the proxy server, which then connects to the target. A mail processor is created as a module object at run time for each couple of source and target sockets. The mail processor parses MIME, reads the buffer from the source socket, scans the buffer, and writes the buffer

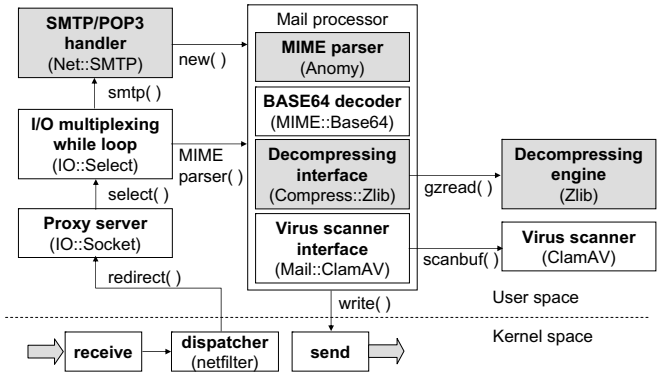

Fig. 3. The software components in the system implementation. The bold texts are module names, and the name in parentheses denotes the open-source package used in that component. A shaded block stands for the package that has been modified for our purposes.

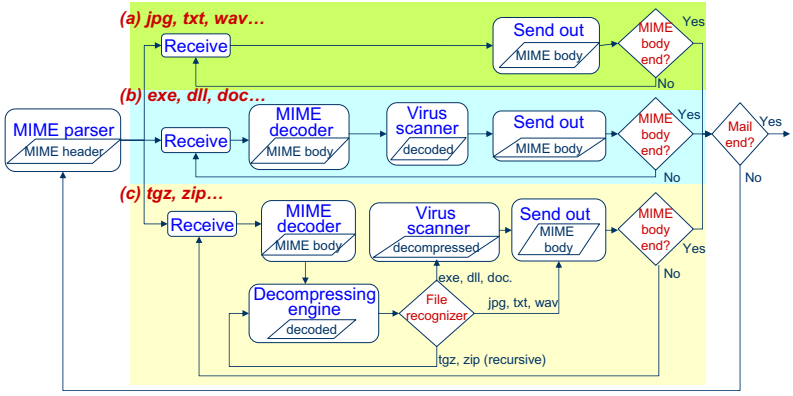

Fig. 2. The flow of processing different mail attachments according to their types.

to the target socket. The MIME parser leverages the code of an open-source mail sanitizer, *Anomy* (http://mail-tools.anomy.net), because it can treat mail as data stream. The mail processor is protocol independent. It can work with the SMTP/POP handler or the handlers of other mail services, say IMAP, in the future work. The packages of *Net::SMTP::Server*, *Compress::Zlib*, *Zlib* and *Anomy* are modified in this architecture. For I/O multiplexing, *Net::SMTP::Server* is modified to process one line at a time when a socket is selected. *Compress::Zlib* is a Perl module to call the *Zlib* shared library. The original *Zlib* will fail if it reads the end of data stream that is not the end of file. We remove this limitation to make partial decompression possible. A mail transport agent, say *Postfix*, is not part of the system because the system is only a virus-scanning mail proxy and it does not need a full set of SMTP functions. These components are well modularized. When the involving packages have new versions, they can be upgraded in this system as long as the arguments of their functions are kept the same.

Fig. 4 presents the interface from *MIME::Base64* to *Zlib*. The current implementation supports only the files compressed by *Zlib*, but it can be extended to support other formats. The original *Zlib* opens a file with the *gzopen* function, and then reads the file with the *gzread* function for decompression until an EOF (end of file) symbol is met. *Compresses::Zlib* drives the read operation of *Zlib*. This system uses the *pipe* interface (Stevens and Rango, 2005) to pass decoded data from *MIME::Base64* to *Zlib*. The handlers on both sides, *handler_in* and *handler_out* should be set to be non-blocking to allow interleaving processing. The check of the EOF symbol is also eliminated. This mechanism can be applicable to all compression libraries originally designed to handle an entire file.

3.3. Single-process concurrency

This proxy runs as a single process and uses I/O multiplexing for concurrency. Because only one process handles all clients in turn, the system should keep the state of every client. Every time when a client is selected to be handled, the system calls the corresponding function according to the state of the client. Fig. 5 is the state-transition diagram of a client. Except that the *SMTP* and *quit_or_next* states are related to the SMTP protocol, the other states are

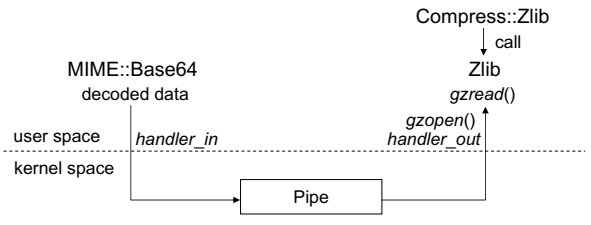

Fig. 4. The interface from *MIME::Base64* to *Zlib* through the *pipe* interface.

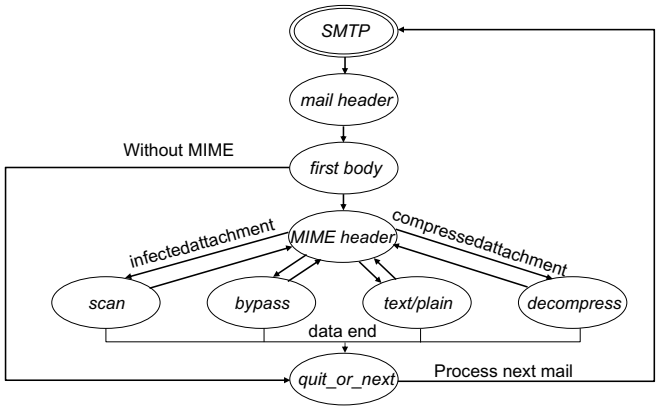

Fig. 5. The states in mail processing for switching between the clients.

MIME parsing states. The states of *bypass*, *scan* and *decompress* handle the attachment in three ways as described in Section 3.1. The state of *text/plain* is used if keyword filtering or anti-spam is part of the mail processing.

The processing time in each state should be short to minimize the latency. The SMTP protocol handler handles one protocol message at a time in the SMTP state. The system reads only 8 kB data each time when handling the attachment. The I/O descriptors in mail processing are all set to be non-blocking by setting the *O_NONBLOCK* flag of the *fcntl( )* function. The non-blocking operations allow fast switching between clients within a single process at the cost of higher complexity in programming. In comparison, the latency in handling an entire mail in a storage-based system forces multiple processes or multi-threading. Both are heavy in Perl implementation.

4. Performance evaluation

4.1. Test bed

We compare the proposed stream-based mail proxy with the storage-based *AMaViS* architecture. Each proxy runs on a Linux (kernel ver. 2.6.10) PC with 1 GHz Pentium III CPU, 512 MB SDRAM and a fast Ethernet card. Perl 5.8.5 runs both proxies since they are implemented in Perl. *ClamAV* 0.83 is the virus scanning engine. *Postfix* serves as the mail transport agent cooperating with *AMaViS* for its full support of *AMaViS*.

For fairness, *AMaViS* is configured in the following ways: (1) disabling the anti-spam function since the stream-based proxy does not check spam, (2) running *ClamAV* in the daemon mode which is faster than the command-line mode and (3) disabling the cache since *AMaViS* bypasses the same mail processed within a configurable period. Both proxies scan the same two mail types in the benchmarks. The first is a mail with a 1 MB executable attachment. The second is a mail with the compressed one from the 1 MB attachment in the first mail (compression rate: 37%).

4.2. The impact of different mail traffic on performance

We measure latency and throughput to compare the performance of both proxies. The mail sender and the target receiver run on the same computer to synchronize the time logged on both the sender and the receiver. The latency is defined to be the elapsed time from the start of sending one mail to the end of receiving on the target. We observe the latency without any proxy, with the stream-based proxy and with the *AMaViS* architecture. Table 2 presents the latency in the three conditions. The stream-based proxy has significantly shorter latency than *AMaViS* in every test condition. We have two more observations. (1) The cost in mail forwarding is much lower in the stream-based proxy than in *AMaViS*. The forwarding cost is heavy in *AMaViS*, even heavier than virus scanning. (2) Even though computation in decompressing is involved for a compressed file, *AMaViS* still has longer latency in handling an uncompressed file than in handling a compressed file. This again indicates the high cost in forwarding a long file. In comparison, forwarding in the stream-based proxy is not a dominant factor, but virus scanning is. A closer look at this observation is presented with the internal benchmark in Section 4.4.

The benefit of the stream-based proxy for ordinary mail whose size is on the order of hundreds of bytes to a few kB is also of concern. We prepared two sets of mail messages for the observation. The first set contains pure text messages. The average message size is around 4 kB. The second set is a mix of 80% of pure text messages (also around 4 kB on average) and 20% of messages with attached executable files (around 35 kB on average). These executable files will be scanned for viruses.

The case for the first set is similar to the processing with only forwarding and other mail processing in Table 2, but we observed that the stream-based proxy is 11.24 times faster than *AMaViS* for the first set, rather than 7.29 times faster in Table 2. The stream-based proxy is more beneficial for pure text messages because storage-based *AMaViS* has fundamental cost in disk operation and in inter-process communications between the daemons of its architecture. The cost is not proportionally reduced with the mail size. For the second set, the stream-based proxy is 3.6 times faster. The value is only slightly higher than 3.48 in the F + V + O case of Table 2. The reason is that virus scanning is a time-consuming part of the processing, and the time spent in virus scanning is similar in both the stream-based proxy and *AMaViS*.

The throughput is defined as the total mail size divided by the elapsed time. A stress test of a large number of identical mails are sent through both proxies to measure the total elapsed time. Because BASE64 encoding expands the file size by a factor of 4/3, we calculate the throughput based on the original attachment size. The throughput of the stream-based proxy on simple forwarding is 65.2 Mb/s, very close to that of 69.93 Mbps without a proxy. *AMaViS* achieves only 9.51 Mbps even though it disables both anti-virus and anti-spam functions, meaning this storage-based proxy itself is a bottleneck even on mail forwarding. Table 3 presents the throughput with virus scanning and decompressing. Virus scanning degrades the throughput of the stream-based proxy from 65.2 Mb/s on mail forwarding to 21.79 Mb/s, meaning virus scanning is a bottleneck. *AMaViS* achieves 6.9 Mb/s with virus scanning, slightly dropped from 9.51 Mb/s on mail forwarding. This result coincides with the observation on the latency that mail forwarding is heavier than virus scanning in *AMaViS*. The throughput for a compressed attachment is measured in two values. The higher one, denoted by appending "_effective", is the effective throughput calculated from the file size after decompression. Because it is the decompressed file to be scanned, the effective throughput represents the real throughput of virus scanning.

4.3. Buffer requirement

We monitor the disk and memory usage of the two proxies with variable number of clients. Each client sends one mail with a 300 kB attachment compressed from a 1 MB file. Fig. 6 presents the space usage of both proxies. The space usage of the *AMaViS* architecture is not scalable because it grows much faster than that of the stream-based one as the number of clients increases. It must be noted that the memory usage is measured in its virtual size. The usage could exceed the size of physical memory, up to nearly 800 MB herein. The use of virtual memory cripples the system further by slower disk accesses.

The space usage in the *AMaViS* architecture come from runtime processes and mail storage. The space of the former is allocated for each process, so it is proportional to the number of clients. The mail storage space is often proportional to both the mail size and the number of clients. In the *AMaViS* architecture, a *Postfix* daemon receives mail from the client and passes it to *AMaViS*. Another *Postfix* daemon receives mail from *AMaViS* and sends it out to

Table 2
Latency of sending a mail with a 1 MB file and the compressed one from the 1 MB attachment. (Notations—F: forwarding, D: decompressing, V: virus scanning, O: other mail processing)

Latency (ms)	No proxy	AMaViS	Stream-based
F + O (original)	102	1553	213
F + O (compressed)	75	780	105
F + V + O (original)	N/A	1802	518
F + D + V + O (compressed)	N/A	1267	527

Table 3
Throughput of the two proxies with virus scanning and decompressing

Average throughput (Mb/s)	AMaViS	Stream-based
Scan	6.90	21.79
Decompress	3.82	8.05
Decompress_effective	10.37	21.86

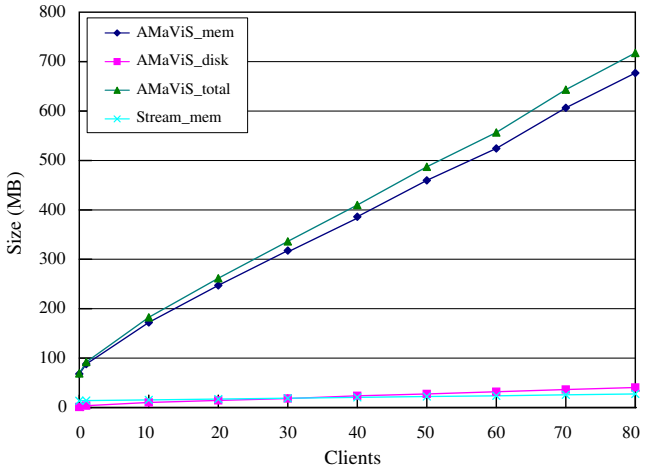

Fig. 6. The space usage of *AMaViS* and the stream-based proxy. AMaViS_mem and AMaViS_disk denote the memory usage and disk usage of the AMaViS architecture, and AMaViS_total is the sum of both. Stream_mem denotes the memory usage of the stream-based proxy, which is diskless in operation, so no disk usage is there.

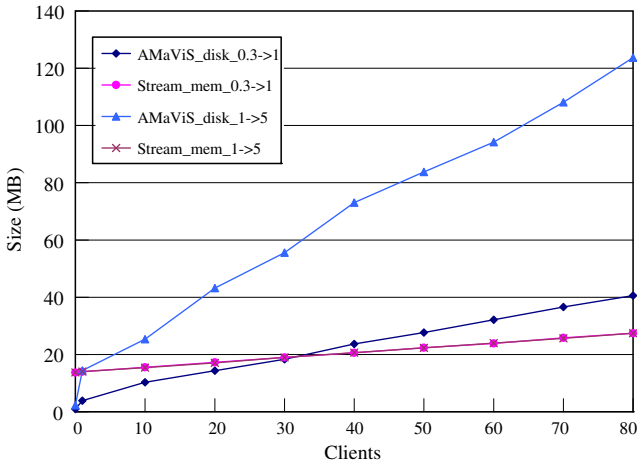

Fig. 7. The space usage of a mail with a 300 kB attachment compressed from a 1 MB file and a mail with a 1 MB attachment compressed from a 5 MB file. AMaViS_disk denotes the disk usage in the *AMaViS* architecture.

the target. The number of *AMaViS* daemons is configurable and is fixed at run time. The number of *Postfix* child processes is the sum of the number of clients and the number of *AMaViS* daemons since the *Postfix* should communicate with both sides. Table 4 lists the names of all related programs, as well as the size and the number of the processes. Let m denotes the number of clients and n denotes the number of *AMaViS* child processes. The total memory usage is

$$(4491 + 2859) * (m + n) + 4259 * 2n + 20430 * n + 19000 + 2759 + 7463.$$

The memory usage grows about 7350 kB per client in the *AMaViS* architecture. This value is the total memory usage by *smtpd* and *cleanup*.

Fig. 7 compares the space usage for different mail sizes. We add another sample mail with a 1 MB attachment compressed from a 5 MB file. The disk usage in storage-based proxy is proportional to the mail size. The memory usage in our proxy remains constant to the mail size because of the streaming operation with interleaving decompressing and virus scanning. The stream-based proxy consumes runtime process space of 13.7 MB in memory when there is no client. The memory usage increases only 176 kB per client for the mail processor that contains the buffers and variables to record the mail states.

Table 5 analyzes the disk usage in the *AMaViS* architecture. *AMaViS* saves a copy of each mail from *Postfix* in its repository, then calls the external program to decompress the files from the original archives, and scans these files for viruses. Let ms_i denotes the size of i-th mail in *Postfix* or *AMaViS*, and $d(ms_i)$ is the size after decompression. The disk usage in *AMaViS* and *Postfix* is $\sum_{i=1}^{m} ms_i + \sum_{i=1}^{n}(ms_i + 2d(ms_i))$.

Table 4
Programs related to *AMaViS* and *Postfix*, where m denotes the number of clients and n denotes the number of *AMaViS* child processes

Program	Description	Size (kB)	Number
smtpd	Postfix SMTP server child process	4491	$m+n$
Cleanup	Process the queue received by smtpd	2859	$m+n$
SMTP	Postfix SMTP sender	4259	$2n$
AMaViS child	AMaViS child process	20,430	n
AMaViS master	AMaViS listening on port 10,025	19,000	1
Postfix master	Postfix listening on port 25 and 10,026	2759	1
Clamd	ClamAV daemon	7463	*1*

4.4. Analysis of internal bottleneck

We use the Perl module *Devel::Profile* (http://www.cpan.org) to record the processing time of each stage in the *AMaViS* architecture and the stream-based proxy. This test uses a mail with an attachment that is 1 MB large after decompression. Table 6 presents the processing time of each stage. The bottleneck in the stream-based proxy is virus scanning, which occupies 67.89% of the execution time. In comparison, the inter-process communications between software components are the dominant factor in

Table 5
Disk usage in *AMaViS* and *Postfix*, where m denotes the number of clients and n denotes the number of *AMaViS* child processes

Program	Description	Size
Postfix	Store all mails on the disk	$\sum_{i=1}^{m} ms_i$
AMaViS	Save the mail being processed	$\sum_{i=1}^{n} ms_i$
	Decompress the file	$\sum_{i=1}^{n} d(ms_i)$
	Copy the files from the archive	$\sum_{i=1}^{n} d(ms_i)$

Table 6
Percentage of the processing time of each stage. The total processing time for AMaViS is 1267 ms, while that for the stream-based proxy is 527 ms

Percentage (%)	Scan	Decompress	Handle MIME	Receive	Send	IPC
AMaViS	25.29	8.78	14.31	8.78	5.53	37.31
Stream-based	67.89	10.78	15.52	3.45	2.36	0

the *AMaViS* architecture. This also explains why the forwarding alone takes much time. Virus scanning is not the primary bottleneck as in the storage-based mail proxy.

5. Conclusion and future work

In this work, we design and evaluate a stream-based mail proxy with interleaving decompression and virus scanning. The file system access is eliminated and the buffer space is saved. The single-process concurrency is made possible with the stream-based design. Compared with the storage-based proxy, the stream-based proxy has the benefits of higher performance, lower latency and economical space usage. The external benchmark shows our proxy has shorter latency and higher throughput in the test conditions. 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% from 69.93 Mbps to 9.51 Mbps in *AMaViS*. The throughput of our proxy is 21.79 Mbps, higher than 6.9 Mbps in *AMaViS* for virus scanning, and it is 8.05 Mbps for decompressing plus scanning, higher than 3.82 Mbps in *AMaViS*. In the space usage, the stream-based proxy grows 176 KB per client in memory while the storage-based proxy grows 7350 KB per client. Besides, the stream-based proxy uses no temporary files on disks, while the disk usage of storage-based proxy is proportional to both the number of clients and the mail size. This system is feasible for an embedded system without a hard disk.

An internal profiling analyzes the bottleneck of both systems. Mail forwarding is the main factor that dominates the processing time in the storage-based proxy, but virus scanning is the main bottleneck in the stream-based system.

For future work, we plan to implement the anti-spam function on the system. Another improvement is implementing the system in C rather than Perl. C is efficient but it takes more effort due to its worse ability of string processing.

Acknowledgement

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.

References

Burrows, M., Wheeler, D.J., 1994. A block-sorting lossless data compression algorithm, Digital SRC Report, no. 124.

Chi, C.H., Deng, J., Lim, Y.H., 1999. Compression Proxy Server: Design and Implementation. USENIX Internet Technologies & Systems, Boulder, CO.

CPSecure, 2006. Stream-based Scanning: CP Secure's Solution, <http://www.cpsecure.com/products/stream-based-scanning.html>.

Freed, N., Borenstein, N., 1996. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies, RFC 2045.

Kermani, P., Kleinrock, L., 1979. Virtual cut-through: a new computer communication switching technique. Computer Networks 3, 267–286.

Liz, 2003. Things you need to know before programming Perl ithreads, <http://www.perlmonks.org/?node=288022>.

Miretskiy, Y., Das, A., Wright, C.P., Zadok, E., 2004. Avfs: An On-Access Anti-Virus File System, the 13th USENIX Security Symposium, San Diego, CA.

Stevens, W.R., Rango, S.A., 2005. Advanced Programming in the UNIX Environment. second ed.. Addison-Wesley, Boston, MA, Chapter 15.

Welch, T.A., 1984. A technique for high-performance data compression. IEEE Computer 17 (6), 8–19.

Wu, K., Yu, P.S., Wolf, J.L., 2001. Segment-based Proxy Caching of Multimedia Streams, World Wide Web (WWW), Hong Kong.

ZDNet India, 2006. ZyXEL and Kaspersky Lab Join Forces to Offer Gateway Antivirus on New ZyWALL UTM. <http://www.zdnetindia.com/news/pr/stories/134119.html>.

Ziv, J., Lempel, A., 1977. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23, 337–342.

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.

Szu-Hao Chen 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 Cyberlink since 2005. His research interests include network security and content networking.

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.

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.