國立交通大學

資訊科學系

碩 士 論 文

一個對於防毒，防廣告信，入侵偵測以及內容過濾的整合性代理伺服器架構

An Integrated Proxy Architecture for Anti-Virus, Anti-Spam, Intrusion Detection and Content Filter

研 究 生：詹智為

指導教授：林盈達 教授

中華民國 九十三 年六月

一個對於防毒、防廣告信、入侵偵測以及內容過濾的整合性代理伺服器架構

學生：詹智為　　　　指導教授：林盈達

國立交通大學資訊科學系

摘要

網路內容安全是一個大眾所關切的重要議題。我們選擇了五個非常普及的開放原始碼套件來解決網路入侵，病毒，廣告信以及不適當的網頁內容等問題。然而光是安裝這些套件會造成四種系統的額外負擔：分別是 process forking、多餘的 IPC、user/kernel space interaction、以及重複的封包重組。為了解決以上的系統額外負擔，我們提出了一個緊密的整合性架構。此架構使用多執行緒以及 *select()*系統呼叫來解決第一種系統負擔。另外此架構整合需要的套件到同一個代理伺服器上來解決第二三四種系統負擔。外部測試顯示系統整合之後，在內容過濾以及入侵偵測的效能從 7.16 Mbps 提升到 13.11 Mbps，在防毒以及防廣告信方面從 2.85 Mbps 提升到 5.82 Mbps。測試結果顯示最大的額外負擔在於 process forking，而內部測試更顯示出在 HTTP 中最大的瓶頸出現在字串比對而在 SMTP 中是檔案系統存取，分別佔 48%以及 62%。最後我們建議幾個方向來改善整個架構，包括字串比對演算法，硬體加速，更多協定支援，以及更多的偵測支援。

關鍵字：網路安全、入侵偵測、防毒、防廣告信、內容過濾器、代理伺服器

An Integrated Proxy Architecture for Anti-Virus, Anti-Spam, Intrusion Detection and Content Filter

Student: Chih-Wei Jan Advisor: Dr. Ying-Dar Lin

Department of Computer and Information Science

Nation Chiao Tung University

Abstract

Network content security has become a critical issue for the Internet. We selected five popular open-source packages to solve the problems of intrusions, viruses, spam, and inappropriate Web pages. However, simply installing these packages brings four kinds of overheads: (1) process forking, (2) redundant IPCs, (3) redundant user/kernel space interactions, and (4) duplicate packet reassembly. To reduce the above overheads, we propose a tightly-integrated architecture. This architecture uses multi-thread and the system call, *select(),* to eliminate the overhead in (1), and is integrated with cooperating packages into a single proxy to eliminate the overheads in (2), (3) and (4). The external benchmark reveals that the improvement of performance is from 7.16 Mbps to 13.11 Mbps in content filtering and intrusion detection, and is from 2.85 Mbps to 5.82 Mbps in anti-virus and anti-spam. It shows that the dominating overhead in the original architecture is process forking. The internal benchmark shows that the main bottlenecks of the content processing are string matching in HTTP and file system access in SMTP, 48% and 62%, respectively. Finally, to scale up this architecture, we suggest directions of improvement, including faster string matching algorithms, hardware accelerators, and more protocol support.

Keywords: *network security, intrusion detection, anti-virus, anti-spam, content filter, proxy*

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Introduction

There are a growing number of intrusions, viruses, spams, and inappropriate Web pages all over the Internet. Detecting and filtering them require content inspection at the application layer, as opposed to traditional packet classification at the network and transport layers. Systems for intrusion detection, anti-virus, anti-spam, and content filtering are available on the market to block these nuisances. These systems used to exist separately in a standalone device. With the increasing computer processing power, it becomes feasible and more economical to integrate more of these functions into a security gateway.

To make the integration compact and efficient, similar components during packet processing and inspection need to be identified and combined. First, packets may need to be reassembled to restore the whole content for easier inspection. Second, the content may be encoded, compressed, and distorted for evasion. It may need to be decoded, decompressed, or normalized before further processing. Third, these applications all need to do some kind of string matching to recognize the signatures.

In this work, we investigate the architecture of a tightly-integrated content security gateway. We selected five popular open-source content applications as the candidates to be integrated into a single gateway. They are Snort [1] for IDS, AMaViS [2]+ClamAV [3] for anti-virus, and AMaViS+SpamAassassin [4] for anti-spam, and DansGuardian [5] for content filtering, respectively. It is possible to simply install these packages in a single system. However, a loosely-integrated architecture inevitably brings out four kinds of overheads: (1) Some applications cannot work alone, but have to work with a cache proxy or a mail server. For example, ClamAV obtains the mail content from AMaViS. Hence there are many inter-process communications (IPC) between these processes. (2) Some components perform identical or similar functions in these applications. They can be combined into one so that there is no redundant processing. (3) Some servers are multi-process servers that fork new processes to serve many clients concurrently. This overhead consumes system resources and slows down context switch. (4) Snort sniffs all incoming packets passing through the network interface. When Web traffic needs to be inspected by Snort for intrusions and by DansGuardian for forbidden URLs or content, it will be copied from the kernel space to the user space twice, one through the packet filter routines, and the other through the normal TCP/IP stack, and then inspected by Snort and DansGuardian, respectively.

The above overheads would result in poor performance. To reduce the above overheads, we propose a tightly-integrated architecture. For the overhead in (1) and (2), we pull up the necessary functions from the cooperated servers and add these functions into our proxy server. Thus there are no redundant inter-process communications in our stand alone proxy server. Identical or similar functions can be combined into one. For the overhead in (3), we modify the proxy server as multi-thread server or use the function call, *select()*, to avoid forking process. For the overhead in (4), we integrated cooperated applications, say Snort and DansGuardian, into one server to avoid duplicate interaction between the kernel and user space.

After integrating these packages, we perform a series of external and internal benchmarks. In external benchmarking, the tightly-integrated architecture is compared with the loosely-integrated one in terms of throughput, latency and number of connections. In internal benchmarking, a detailed profiling of the integrated packages is performed to examine the time spent in each main component. We want to answer the following questions:

(1) What is the influence on the performance of each feature in these packages?

(2) Where are the bottlenecks of these modules?

(3) How much can the tightly-integrated architecture improve the performance?

The rest of this work is organized as follows. Chapter 2 gives a brief survey of an integrated architecture in both research studies and commercial products. We then describe the candidate open-source packages and inspect the packet flow. Chapter 3 presents our integrated architecture and its implementation. Chapter 4 shows the performance gain by benchmarking both internally and externally. Finally, the study is concluded in Chapter 5.

Chapter 2 The Integrated Architecture

2.1 Related Works

The early network firewalls mostly emphasized on providing robust security and preserving high performance [6][7]. After that, the research started to integrate more secure functions into the network firewall [8]. Recently, as needs in security diversification, more security systems were integrated into one.

For improving proxy performance, little research has been done on topic development in improvement of TCP splicing for application proxy performance with kernel support [9][10]. There is also study in reducing overheads to minimize system costs [11].

Typical network security devices include firewalls, VPN devices, IDSs, anti-virus systems, anti-spam systems, and content filtering systems. Firewalls and VPN devices usually exist in a single box [12][13][14][15]. Content security devices, such as IDSs, anti-virus systems, anti-spam systems, and content filtering systems, usually exist in a separate box each [16][17][18]. These devices either operate standalone or receive redirected traffic from firewalls.

As the computer processing power increases, there is a tendency to integrate more functions into one box. Commercial products, such as Symantec Gateway [19], FortiNet [20], Astaro Security Linux [21], integrate all of the above functions in a box. How to integrate many functions efficiently becomes a practical issue in both research field and market place [22][23].

2.2 Selected Packages

As opposed to commercial products, we select open-source packages for easier observation because the source code is available. The selected software packages are introduced herein. Only Snort can work alone. The others need to cooperate with a cache proxy server or a mail server. The packet flow of our integration is explained in detail in the next section.

TABLE 1 Open-source packages.

Package name	**Type**	**Language**	**Interface**	**Version**
Snort	Single process	C	BPF	2.0
AMaViS	Multi process	Perl	Socket	20030616-p9
ClamAV	Single process	C	Socket	0.7
SpamAssassin	Plug-in	Perl	Function call	2.6.3
DansGuardian	Multi process	C++	Socket	2.0

Table 1 lists five open-source packages we selected for integration. Here Snort is an intrusion detection system. ClamAV and SpamAssassin, working with AMaViS and a mail server, serve as anti-virus and anti-spam systems, respetively. In this work, we chose Postfix for the mail server because of its complete support with AMaViS. The system of content filter consists of DansGuardian and a cooperated cache proxy server, Squid.

Figure 1 shows the components and packet flows of the selected packages. First, Snort sniffs packets with libcap, and then does a series of preprocessing, de-fragmentation, re-assembly, normalization, etc. Next, the packets are sent to the detection engine for signature matching. Finally, if any intrusion is found, Snort will generate alerts and may log the packets for further analysis.

AMaViS receives mail from the MTA. Upon receiving the mail, AMaViS proceeds with the MIME Handler to check whether there are any attached files. If there are any, AMaViS would recognize the file types and decompress them if they are compressed files. Last, AMaViS sends decoded text to SpamAssassin for anti-spam check and decompressed files to ClamAV for anti-virus scan. It decides to block this mail or not according to the results from SpamAssassin and ClamAV.

After receiving the message from AMaViS, SpamAssassin starts to read the message and match against a list of signatures to determine whether it is a spam or not. Finally, it returns the result to AMaViS to decide whether if this mail is a spam. ClamAV is invoked by AMaViS when anti-virus is needed. It reports whether there is any virus in the files.

DansGuardian receives a request from the client, and then proceeds with a series of IP address and URL checking. If the request is permitted, DansGuardian passes the request to Squid, or block it directly.

2.3 Complete Packet Flows

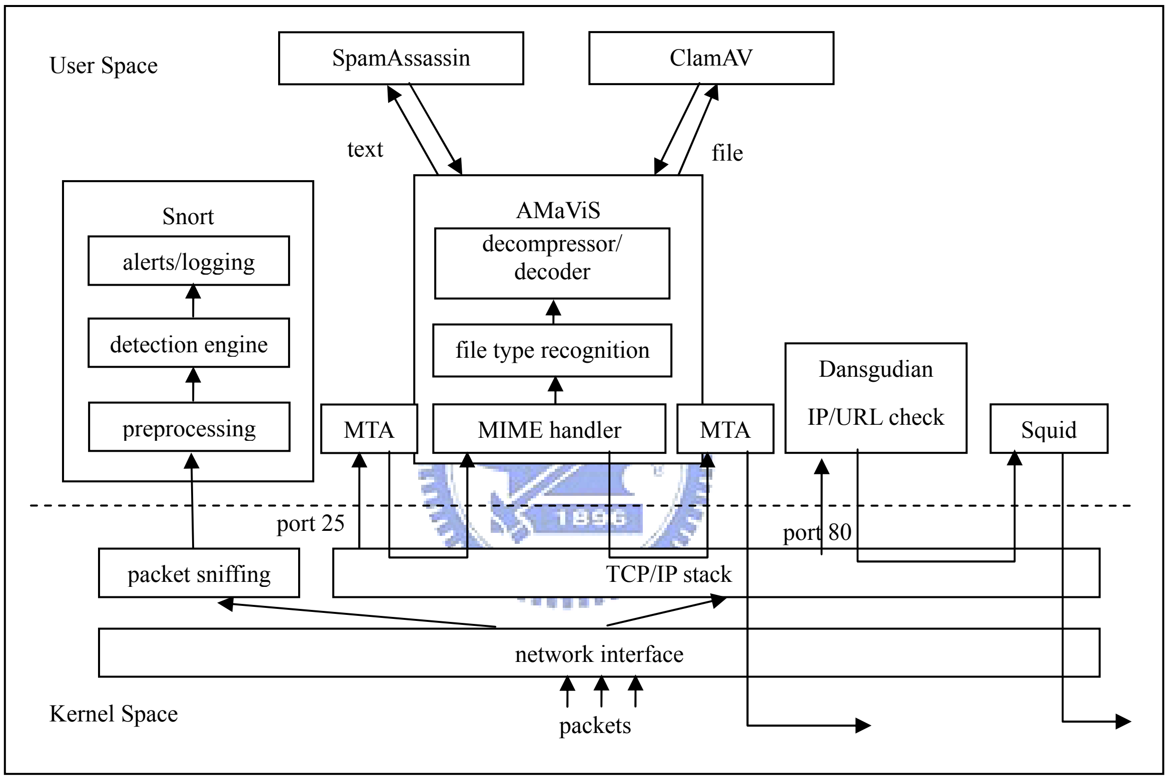

FIGURE 1 Components and packet flows of a loosely-coupled architecture.

Now let us examine the detailed HTTP and SMTP packet processing flows of the integrated security gateway. In the HTTP part, if both IDS and Web filtering functions are turned on, the Web traffic flow is as follows:

(1) The packets are sniffed by Snort and copied from kernel to user space.

(2) The request is also received by DansGuardian through the TCP/IP protocol stack.

(3) DansGuardian checks whether the URL of the request is permitted to access.

(4) If permitted, the request is received by Squid.

(5) Squid makes a connection to the Web server if necessary.

In this Web processing flow, there are *three* user/kernel interactions in steps (1), (2), and (4), and *one* inter-process communication in step (4).

In the SMTP part, the mail traffic flow is as follows:

(1) The packets are sniffed by Snort and copied from kernel to user space.

(2) The kernel passes the mail to localhost:25.

(3) The mail is received by the MTA through the TCP/IP protocol stack.

(4) The MTA forwards mail to AMaViS.

(5) AMaViS calls SpamAssassin to check mail.

(6) AMaViS sends message to ClamAV to scan files attached in the mail.

(7) AMaViS forwards mail back to the MTA.

(8) The MTA relays mail to mail server.

In this mail processing flow, there are *four* user/kernel interactions in steps (1), (3), (4), and (7), *two* inter-process communications in steps (4) and (7), *one* process invoking in step (6), and *one* file system access in step (4).

Obviously, there are many redundant user/kernel interactions and inter-process communications. This consumes more system resources and time. Furthermore, some servers fork processes to serve many clients concurrently. Context switch between these processes take more system resources. These overheads reduce system performance.

Chapter 3 Design of a Tightly-coupled Architecture

3.1 Solution Ideas

As mentioned in Chapter 2, a loosely integrated architecture would bring many overheads during packet processing: inter-process communications between different processes, duplicate kernel/user space interactions made by different operations in different applications, and process forking for serving many clients concurrently.

In this chapter, we propose a tightly integrated architecture to reduce the above overheads. First, in order to reduce redundant inter-process communications, we design our new architecture as a standalone proxy server without cooperated server support. Second, we integrate applications that deal with the same application protocol into one. Therefore, duplicate kernel/user space interactions in different operations will be eliminated. Third, the architecture is modified as a single process proxy server. We use the following two solutions to serve many clients concurrently. The first is using the *select()* system call, which examines the I/O descriptor sets to see if any descriptors in the sets are ready for reading, writing, or have exceptional condition pending. Hence, we can do I/O multiplexing on many socket descriptors to serve many clients concurrently. The second is multi-threading. Threads are more light-weighted processes and are expected to serve more clients [24].

3.2 The New Architecture

Figure 2 shows our tightly integrated architecture and traffic flow. We separate our architecture into two parts: HTTP and SMTP. In the HTTP part, we replace DansGuardian with a self-developed Web proxy server, Webfd, which has simple URL filtering and keyword blocking functions. It uses the *select()* system call to serve many clients concurrently, unlike DansGuardian which forks new processes to server new clients. The *select()* system call is more scalable and efficient than context switch between processes. To better filter Web content, we also supplement Webfd with the content filtering part extracted from

DansGuardian.

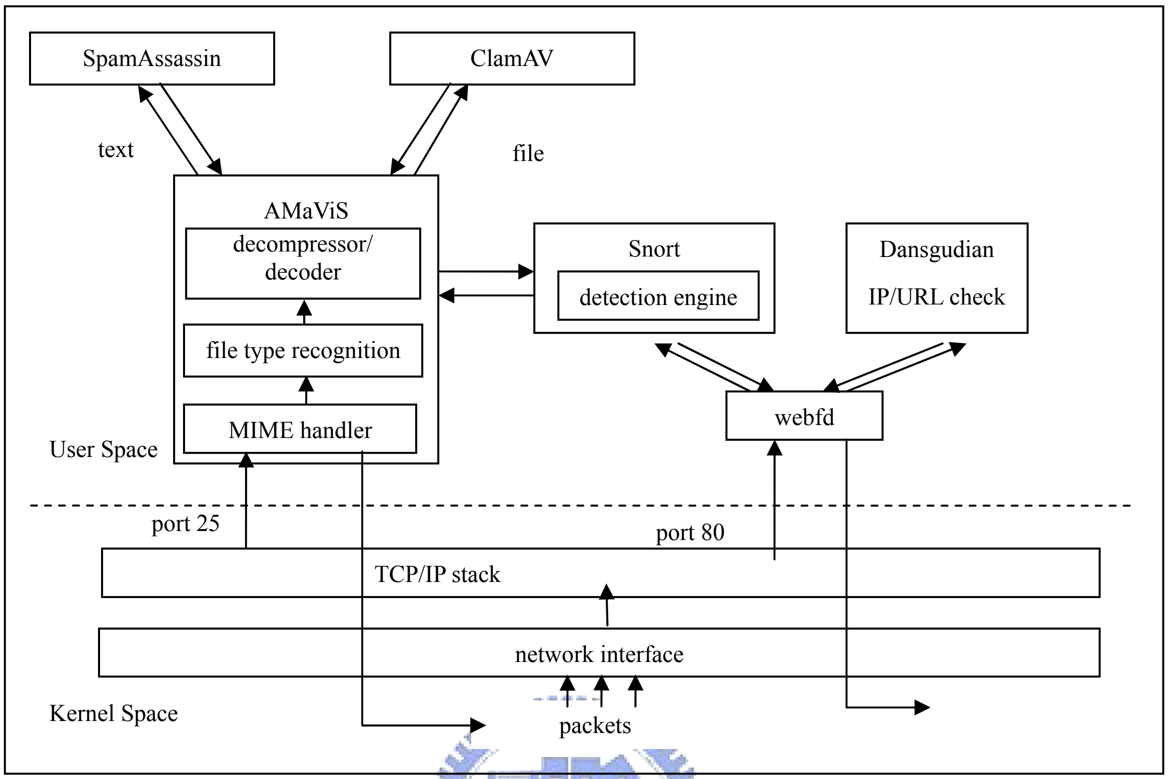

FIGURE 2 Our integration architecture and the new packet flows.

On the other hand, to reduce duplicate kernel/user space interactions and to provide intrusion prevention instead of intrusion detection, we rewrite Snort as a shared library called by Webfd to detect or prevent intrusions. Snort originally sniffs packets with libcap that copies packets passing through the network interface from kernel to user space. Snort can only detect the intrusion on the network link. Because Webfd is a proxy, after receiving reassembled packets from the TCP/IP protocol stack, it calls the rewritten Snort shared library to detect possible intrusions. If an intrusion is found, the packets can be blocked. Making Snort as a shared library allows proxies of other protocols to call it to detect and prevent intrusions. Any proxy could fill the data structure *Packet* in the Snort library and start the detection engine. Furthermore, there is only one copy of Snort residing in memory.

In our architecture, the intrusion detection and prevention of a new protocol has to rely on the proxy of the protocol that extracts the content to be inspected. However, it is not worth running a new proxy if we just want to detect the intrusion through such a protocol. Snort originally uses libcap that offers a simple way to sniff packets, an easier and more efficient approach for intrusion detection. We hence can just run Snort in sniffing mode if we only want to detect the intrusions. After our integration, the Web traffic flow in a gateway form the client side to the server side becomes as follows:

(1) The kernel passes packets to localhost:880.

(2) The request is received by Webfd through the TCP/IP protocol stack.

(3) Webfd checks whether the URL of the request is permitted to access.

(4) Webfd calls Snort library to check whether the content of the request contains intrusion signatures.

(5) Webfd makes a connection to the Web server if necessary.

In the SMTP part, we modify AMaViS as a standalone mail proxy server. Furthermore, to be more scalable, we alter AMaViS from a multi-process proxy server to a multi-thread proxy server. We adopt multi-thread instead of the system call, *select()*, in the mail proxy server because the processing time of the mail is much longer than the processing times of the Web request and response. In our benchmarking, the processing times of the Web request and response are at most tens and hundreds of milliseconds, respectively. However, the process time of the mail is hundreds to several thousands of milliseconds. This would degrade the concurrency of the mail proxy server. The mail proxy server hence is modified as a multi-thread proxy server. We also run ClamAV in the daemon mode to save the time of loading ClamAV and its signatures. AMaViS hence communicates with ClamAV daemon by socket. The original SpamAssassin is still as a PERL library residing in the memory. AMaViS also calls the rewritten Snort shared library to detect possible intrusions. After the integration, the mail traffic flow in a gateway from the client side to the server side becomes as follows:

(1) The kernel passes packets to localhost:10024.

(2) The mail is received by AMaViS through the TCP/IP protocol stack.

(3) AMaViS calls Snort library to check whether the content of the mail contains intrusion signatures.

(4) AMaViS calls SpamAssassin to check mail.

(5) AMaViS sends message to ClamAV to scan the attached files.

(6) AMaViS relays the mail to the next mail server.

After the integration, the redundant inter-process communications and the duplicate user/kernel space interactions are eliminated. Comparing to the old architecture, there are totally *two* user/kernel space interactions in steps (2) of both parts, *one* inter-process communications in step (5) of the SMTP part, and *one* file system access in step (2) of the SMTP part. Furthermore, there are only *two* server processes. The improvement of the integration will be shown in the next chapter.

3.3 Implementation

There are three major changes in the new implementation: standalone AMaVis, Snort as a shared library, and Webfd.

3.3.1 Standalone AMaViS

In AMaViS, there are two important modifications that make AMaViS a multi-threaded and standalone server. By standalone, we mean that AMaViS can serve clients without the mail server support. First, AMaViS originally uses the Net::Server::PreForkSimple module. The Net::Server [25] is an extensible, generic Perl server engine. We hence can easily extend the Net::Server or modify its sub classes directly and replace Net::Server::PreForkSimple with it. We use the threads [26] module, a new module in Perl 5.8 to implement the multi-threaded server. We modified the subroutine *loop()* in the Net::Server::Fork module. In this subroutine, it forks a process to serve a new client when a request is coming. We replace this piece of code with the one that creates a new thread. Second, AMaViS originally forwards the checked mail to the local mail server in the subroutine mail_via_smtp_single of

Amavis::Out. We hence modify this piece of code to relay the mail to the destined mail server.

3.3.2 Snort as a Shared Library

To make Snort a shared library, we compile each source file with the option –fPIC, and finally use the command, ld –share, to make this shared library. Next, in order to do intrusion detection or prevention with Snort, we call *fpInitDetectionEngine()* to initialize the detection engine and *CreateDefaultRules()* to create default rules at the beginning of Webfd. When a request or response arrives, the data structure of *Packet* is filled with the content of the request or response, and then the function *fpEvalPacket()* will analyze *Packet*. Last, the detection result will be returned to the caller, i.e. Webfd or AMaViS in this case.

3.3.3 Webfd with the Features of DansGuardian

Part of DansGuardian is compiled as a library called by Webfd. The request inspection of DansGuardian is in the class *OptionContainer*. The inspection to check whether the specified keywords are in the URL by regular expression is in the function *inBannedRegExpURL()*. The inspection to check whether the URL and the site are permitted are in the functions *inBannedURLLIst()* and *inBannedSiteList()*, respectively. Webfd calls these functions sequentially when processing the request. On the other hand, the response inspection of DansGuardian is in the class *NaughtyFilter*. The inspection to check whether the statistical score of the specified keywords appearing in the response exceeds the threshold is in the function *checkme()*. Similarly, Webfd calls this function when processing the response.

Chapter 4 Benchmarking

4.1 External Benchmarking

In this chapter, we compare the performance of our new tightly-coupled architecture with the loosely-coupled one. We install both solutions on a PC with Pentium!!! 1G CPU, 256MB SDRAM and 20GB hard disk.

TABLE 2 Categories of external benchmark.

Category	**Benchmark Tool**	**Benchmark Item**	**Performance Metrics**
HTTP test	WebBench	1. Influence of Snort 2. Influence of response size	Throughput
SMTP test	Modified Postal	1. Influence of anti-virus 2. Influence of anti-spam	Throughput

Table 2 describes the benchmark tools, benchmark items, and performance metrics. WebBench [27] can send as many requests as possible per second to Web servers to measure the maximum requests per second and maximum throughput in bytes per second. On the other hand, our mail client, the modified Postal [28], can deliver as fast as it can to mail servers so that it can be used to measure the maximum mails per second and the maximum throughput in bytes per second.

In the HTTP benchmark, an Apache 1.3.12 Web server is set up on one side of the gateway, and WebBench acts as Web clients on the other side. There are eight PCs to simulate Web clients. Each PC simulates 10 clients. The Web pages obtained from the WebBench package do not contain any ActiveX/Java objects. Therefore, the gateway only checks whether the Web page is permitted to access by inspecting the content against the keyword database. The benchmark helps to understand not only the difference between the old and new architectures, and also the influence of the functions including URL blocking, keyword filtering, and intrusion detection.

In the SMTP benchmark, like the HTTP benchmark, a mail server is set up on one side of the gateway, and we use the modified Postal acting as mail clients on the other side. The original Postal is used to benchmark SMTP servers. It can only generate mail messages of random data, but does not allow specifying attached files. Instead, the modified Postal allows attached files in the mail. The mails sent from clients to the server contain one line text and a 1 MB file. The gateway then checks whether the mail is a spam or contains a virus. This benchmark helps to understand the influence of the functions including anti-spam and anti-virus.

4.2 Performance Results

4.2.1 HTTP test

Content filtering requires the content to be scanned, and can thus be sensitive to both content count and content size. It scans the URL in the request, and also filters the content in the response. The size of a Web page is typically from 4 KB to 8 KB and does not exceed 40 KB. We hence use the 5KB and 40KB Web pages for benchmarking.

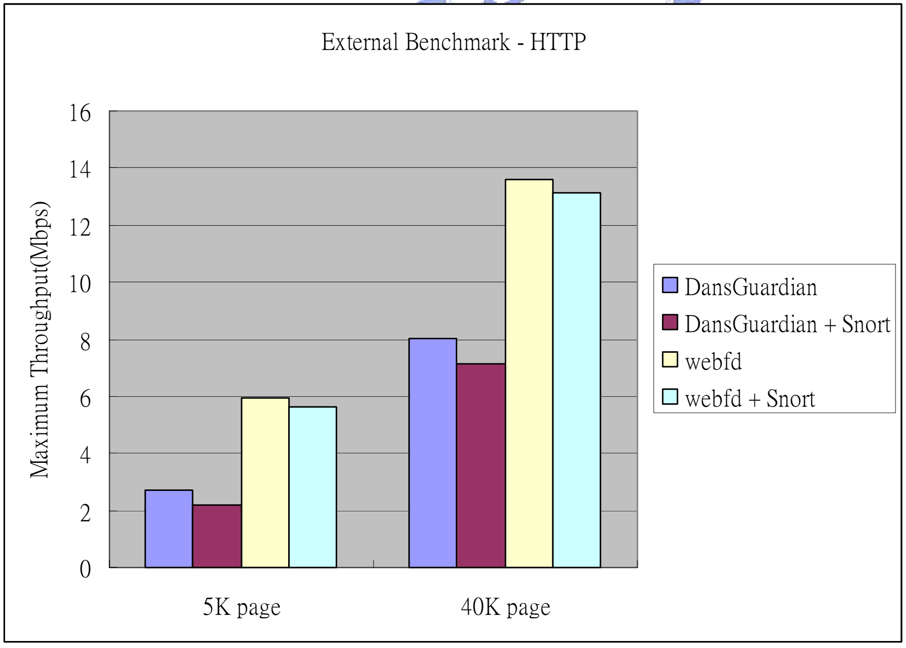

FIGURE 3 Maximum Throughput with or without Snort.

Figure 3 compares the maximum throughput of DansGuardian and Webfd, with or without the IDS, Snort. Webfd improves the maximum throughput by 70% of DansGuardian in the case of 40K page, and over doubled in the case of 5K page. This is due to that Webfd uses the system call, *select()*, without forking processes. On the other hand, the influence of Snort in degrading the maximum throughput in Webfd is smaller than that in DansGuardian, by 0.48Mbps and 0.86 Mbps, respectively.

4.2.2 SMTP test

AMaViS examines the incoming mail of MIME type to properly decode the content and decompress the attached files. Following the decoding stage, SpamAssassin checks the header and the content of the mail, and ClamAV scans the attached files.

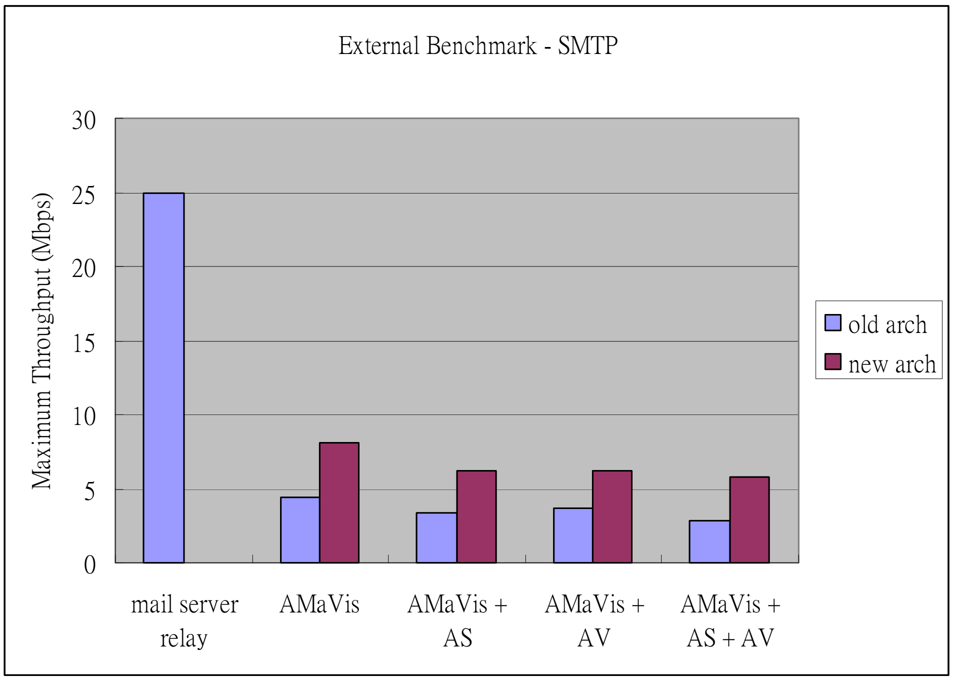

FIGURE 4 Maximum Throughput.

Figure 4 compares the maximum throughput of the old and new architectures, with or without the functions of anti-spam and anti-virus. When only the mail server relays the mails, the maximum throughput arrives at 25 Mbps. In comparison, if the mails are passed through both the mail server and AMaViS, without enabling the functions of anti-spam and anti-virus, the maximum throughput is significantly degraded down to 4.4 Mbps. It consumes more system resources and time because AMaViS forks processes to serve clients and execute external programs. After we modified the architecture to be a multi-thread server, the maximum throughput in every configuration performs better, doubling the throughput of the old architecture. Nevertheless, AMaViS involves the file system access, decoding and decompressing processing. The throughput of AMaViS is much lower than the mail server.

4.3 Internal Benchmarking

To further identify the bottlenecks of the open-source solutions and our integrated architecture, we conduct a series of internal benchmark experiments, as shown in Table 3.

TABLE 3 Categories of internal benchmark.

Category	**Benchmark Tool**	**Settings**	**Benchmark Items**
HTTP test	Chariot gprof fragroute WebBench	1. Enable preprocessors 2. HTTP traffic 3. Enable all IP/URL check functions	Who tops all processing time among all the functions
SMTP test	Modified Postal	1. Enable anti-virus 2. Enable anti-spam	Who tops all processing time among all the functions

WebBench is used to generate HTTP requests to retrieve specific Web pages of different sizes. The time-stamps are taken by the gettimeofday() system call before and after the code segments with an accuracy of microsecond. On the other hand, we also use gprof [29] to collect information during the execution of the program. gprof generates a profile that shows how much time the program spends in each function, and how many times the functions are called. In this test, Chariot [30] is used to generate the real traffic, and fragroute [31] is used to fragment the packets to enforce Snort to de-fragment and reassemble them. In the SMTP benchmark, the modified Postal is used to send mail attached with the specified files. The time-stamps are taken by the build-in log system of the AMaViS with an accuracy of millisecond. UNIX tools, ps and top, are used to examine the memory and disk utilization.

4.4 Resources Consumption

4.4.1 HTTP test

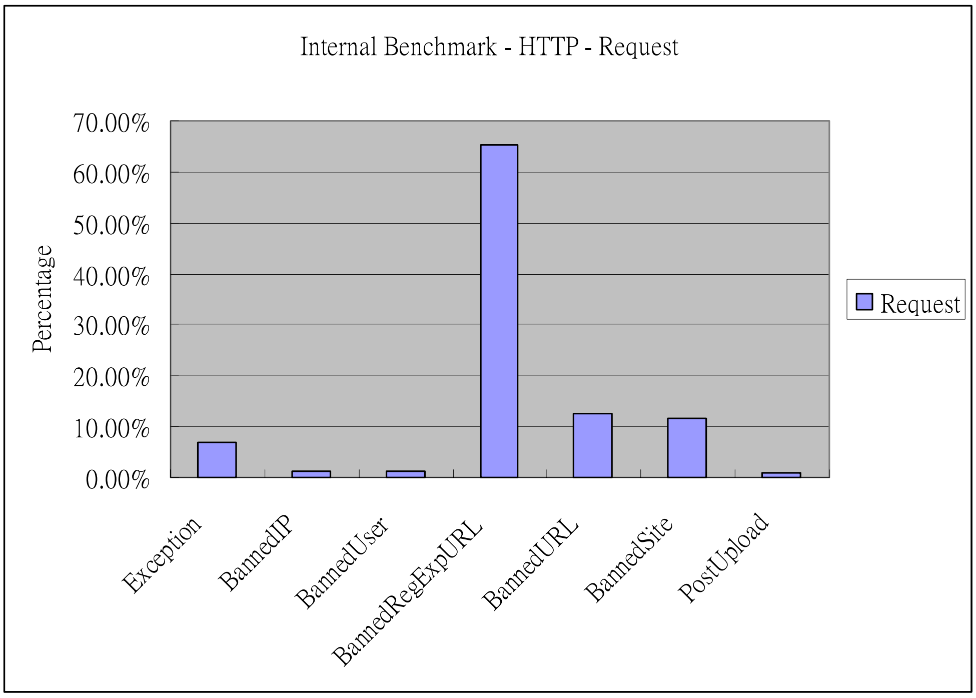

FIGURE 5 Delay Percentage in Request Processing.

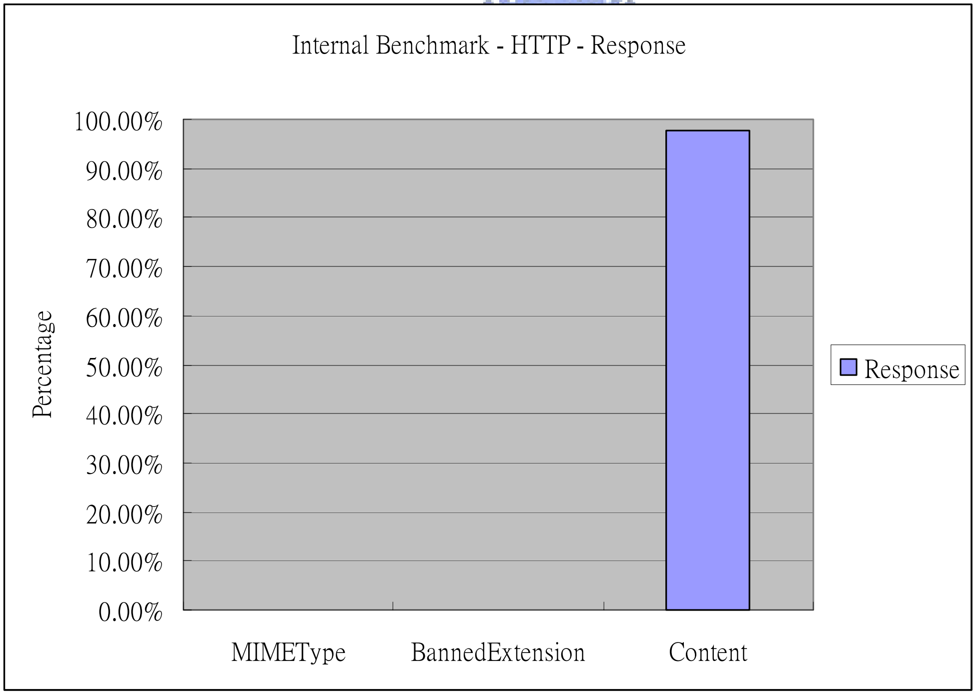

FIGURE 6 Delay Percentage in Response Processing.

Figure 5 and Figure 6 show the delay percentage in the request processing and response processing respectively. In the request process, the term "BannedRegExpURL" means the inspection to check whether the specified keywords are in the URL of the request by using the regular expression, and the terms "BannedURL" and "BannedSite" mean the inspection to check whether the URL or site is in the URL database or site database respectively. The BannedRegExpURL dominates more than 60% when processing the request. It appears that the string matching is the most time consuming of the whole process. However, the main inspections of the request processing are BannedURL and BannedSite which account for totally 24% only. The keyword database of BannedRegExpURL is much small than the URL databases of the BannedURL and BannedSite. However, the processing time of BannedRegExpURL is longer than the processing times of BannedURL and BannedSite. This is due to that the databases of BannedURL and BannedSite can be formed trees. The processes of BannedURL and BannedSite are hence faster than the process of BannedRegExpURL. In our testing, the effect of BannedRegExpURL to block the forbidden request is not obvious. We hence recommend turning off this inspection. In the response processing, the term “Content” means the inspection to check whether the statistical score of the specified keywords appeared in the content of the response exceeds the threshold, and the terms of “MIMEType” and “BannedExtension” are the inspections to check the part of MIME and file extension respectively. The Content with the percentage over 90% is the most time consuming inspection. This is consistent with the earlier finding suggesting that the string matching dominates the whole process.

Figure 7 shows the percentage of the Snort detection process. The String Matching forms a large proportion of the whole process. It is even over 45% in the settings of 40 and 80 bytes. However, since the number of sessions is identical, the proportions of Session Finding and Rule Operation decrease in the configurations of 40 and 80 bytes. On the other hand, the proportion of TCP reassembly is as large as we thought previously. It hence is not a bottleneck of the detection process.

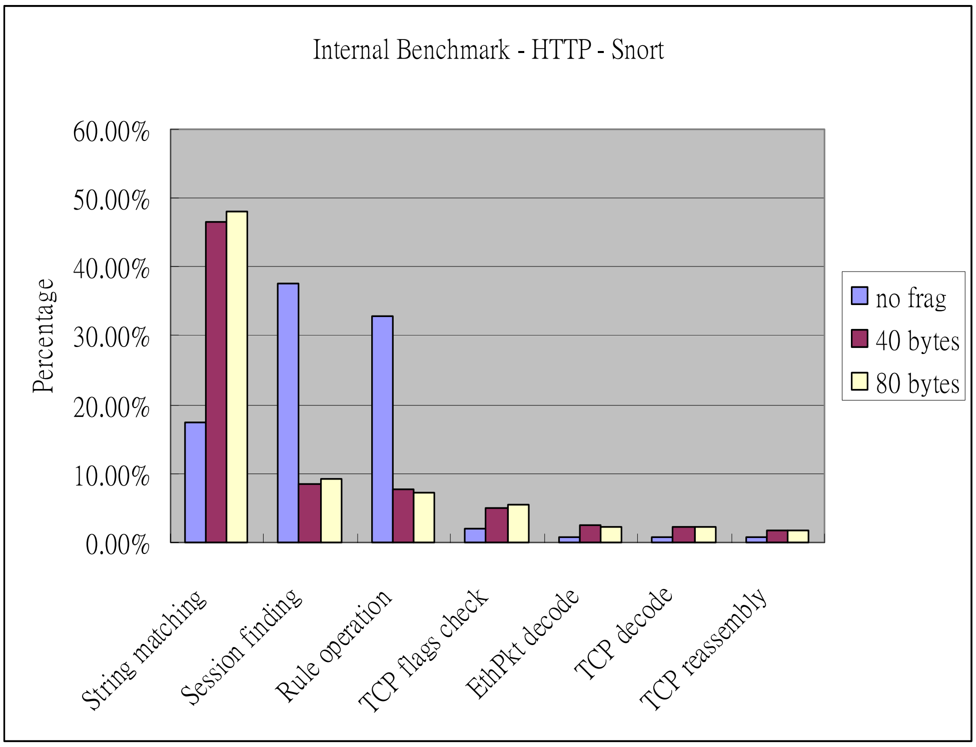

FIGURE 7 Delay Percentage in Snort.

4.4.2 SMTP test

Figure 8 displays the latency of each component in the SMTP traffic processing. In this test, the term "AMaViS" means the processing of the AMaViS, which includes decoding of the content and decomposing of the files. Similarly, the term "Avscan" means the process of invoking the external scan engine to scan the files in the specific directory. In the new architecture, using the scan daemon residing in the memory, instead of invoking external program significantly, reduces the processing time of file scanning. It is only 1/9 of the old architecture. Moreover, making AMaViS a standalone server without the mail server support reduces the communication between AMaViS and the mail server. The processing in the mail server, including Mail receive, Enqueue and Mail send, is hence omitted. However, omitting these overheads raises the processing time of AMaViS in SMTP receive and SMTP forward, 21% and 62%, respectively. These processes involved with file system access hence become the bottlenecks in the new architecture.

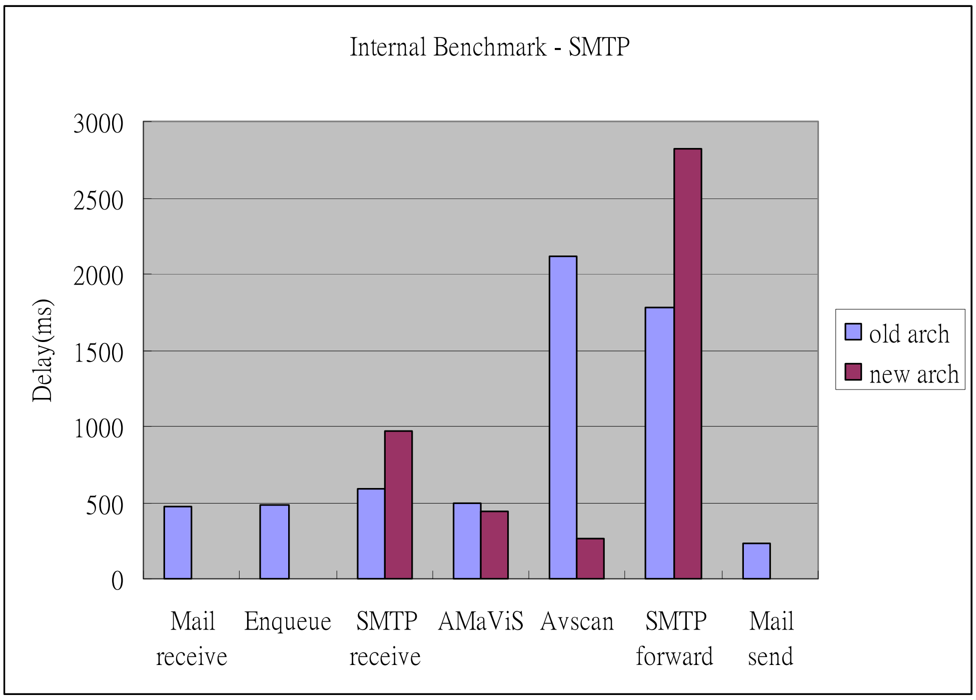

FIGURE 8 Delay in SMTP test.

Since the content size of the mail is much larger than the request, the mail needs to be saved in the disk before being processed. The remaining processings, decoding, decompressing, virus scanning, and spam checking, all involve file system access. The throughput in SMTP hence is much lower than the throughput in HTTP.

Chapter 5 Conclusions and Future Research

This work presents not only the experiences of integrating many open-source packages into a content security gateway, but also the way to further integrating these packages tightly. The overheads among the software components and kernel are reduced. The external benchmarking compares the tightly integrated architecture with the old architecture. The internal benchmarking examines the delay of each key component of the integration.

The external benchmarks have shown that the modification of content filter proxy server from process forking to the system call, *select()*, improves the throughput by 70% ~ 120%. The system of anti-virus and anti-spam is also modified from process forking to multi-threading, doubling the throughput of the old architecture. Therefore, the dominant overhead in the old architecture is process forking. The internal benchmarks show that the content processing bottlenecks of DansGuardian and Snort are string matching. The string matching operation accounts for 65% in the request processing and 97% in the response processing of DansGuardian, respectively. In Snort, it is 17% ~ 46%. On the other hand, file system access is the most time consuming in AMaViS. The file system access operation accounts for 21% in the SMTP receive processing and 62% in the SMTP forward processing.

Since the string matching of content processing is the main bottleneck, to scale up the process, the string matching operations can be offloaded to an accelerator or ASIC. Typical operations are linear string matching algorithm, and multi-pattern string matching algorithm.

On the other hand, the file system access in AMaViS also results in poor performance. To improve this, we can create a RAM disk or modify AMaViS to support streaming operation. After that, the processings of AMaViS can operate the content of the mail in memory without file system access.

Furthermore, in order to enhance these packages, there can be two more improvements. First, in the system of anti-virus and anti-spam, AMaViS only supports the protocol of SMTP.

If the mails are received through the protocol of POP3 or IMAP, these mails would not be scanned. We hence can enhance the AMaViS to support the protocols of POP3 and IMAP, or use the POP3 and IMAP proxies to support the systems of anti-virus and anti-spam directly. Second, in our architecture, the number of protocols with intrusion prevention depends on the proxies we include. If we would like to prevent more protocols from intrusions, we can run more proxies, FTP proxy, for example, on our security gateway to support the intrusion prevention precisely.

References

[1] Snort, http://www.snort.org/ .

[2] AMaViS, http://www.ijs.si/software/amavisd/ .

[3] ClamAV, http://www.clamav.net/ .

[4] SpamAssassin, http://www.spamassassin.org/index.html .

[5] DansGuardian, http://dansguardian.org/ .

[6] N. A. Noureldien, and I. M. Osman, "*A Stateful Inspection Module Architecture*," TENCON 2000, vol. 2, pp. 259-265, September 2000.

[7] P. Gupta, and N. McKeown, "*Algorithms for Packet Classification*," IEEE Network, vol. 15, issue 2, pp. 24-32, March-April 2001.

[8] Y. Lin, H. Wei, and S. Yu, "*Building an Integrated Security Gateway: Mechanisms, Performance Evaluations, Implementations, and Research Issues*," IEEE Communications Surveys, December 2002.

[9] O. Spatscheck, J, Hansen, J, Hartman, and L, Peterson, "*Optimizing TCP forwarder performance,"*, Transactions on Networking, 8(2):146--157. IEEE; ACM, April 2000.

[10] D. A. Maltz and P. Bhagwat, "*TCP Splicing for Application Layer Proxy Performance*," IBM Research Report RC 21139, March 1998.

[11] D. C. Schmidt, T. Harrison, and N. Pryce, "*Thread-Specific Storage -- An Object Behavioral Pattern for Accessing perThread State Efficiently*," in The 4 Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-34), September 1997.

[12] Cisco, http://www.cisco.com .

[13] WatchGuard, http://www.watchguard.com .

[14] NetScreen, http://www.netscreen.com .

[15] Check Point, http://www.checkpoint.com .

[16] ISS, http://www.iss.net .

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

[18] WebSense, http://www.websense.com .

[19] Symantec, http://www.symantec.com/index.htm .

[20] FortiNet, http://www.fortinet.com .

[21] Astaro Security Linux, http://www.astaro.com/ .

[22] C. Ding, C. Chi, J. Deng, and C. Dong, "*Centralized Content-Based Web Filtering and Blocking: How Far Can It Go?*," IEEE Transaction on System, Man, and Cybernetics, vol. 2, pp. 115-119, October 1999.

[23] R. Knobbe, A. Purtell, and S. Schwab, "*Advanced Security Proxies: An Architecture and implementation for High-Performance Network Firewalls*," Proceedings of the DARPA Information Survivability Conference and Exposition, vol. 1, pp. 140-148, January 2000.

[24] A. Tanenbaum, *Modern Operating Systems*, pp. 508-511, Prentice-Hall, 1996.

[25] Net::Server, http://search.cpan.org/~bbb/Net-Server-0.87/lib/Net/Server.pm .

[26] threads, http://search.cpan.org/~nwclark/perl-5.8.4/ext/threads/threads.pm .

[27] WebBench, http://www.veritest.com/benchmarks/webbench/default.asp .

[28] Postal, http://www.coker.com.au/postal/ .

[29] GNU gprof, http://sources.redhat.com/binutils/docs-2.12/gprof.info/ .

[30] Chariot, http://www.netiq.com/products/chr/default.asp .

[31] fragroute, http://monkey.org/~dugsong/fragroute/ .