國立交通大學

資訊科學系

碩 士 論 文

容 錯 代 理 伺 服 器 之 設 計 與 實 作

The Design and Implementation of Fault Tolerant Proxy Server

研 究 生：吳家志

指導教授：張瑞川　教授

中 華 民 國 九 十 三 年 六 月

容錯代理伺服器之設計與實作

The Design and Implementation of Fault Tolerant Proxy Server

研　究　生：吳家志	Student：Chia-Chih Wu
指導教授：張瑞川	Advisor：Prof. Ruei-Chuan Chang

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 2004

Hsinchu, Taiwan, Republic of China

中華民國九十三六年六月

容錯代理伺服器之設計與實作

研究生：吳家志　　　　　　指導教授：張瑞川教授

國立交通大學資訊科學所

論　文　摘　要

代理快取技術是一種被廣泛使用於增進網路使用效率及降低存取時間的技術。然而，研究指出仍然存在著相當多的原因會使得系統中斷服務。一旦系統中斷服務，使用代理主機存取網站資料的用戶將因此無法獲得正常服務。他們必須等待系統管理者發現並修復系統的錯誤。儘管已有很多技術能夠讓代理主機服務能夠容在有錯誤的情況下運作，但是那些當代理主機錯誤時正在服務的用戶請求，卻無法被回復。

我們提出了一個容錯的代理主機系統。他能夠在用戶端以及網頁伺服器端沒有察覺的情況下，發覺本身的錯誤，並且回復所有在錯誤發生時正在服務的請求。我們採用 FT-TCP 的技術來修復連線。這個系統建構在作業系統層級，透過攔截 TCP 通訊協定的封包以及一些系統呼叫（system call）來紀錄代理主機軟體的狀態，並且在它發生錯誤時回復它的狀態。根據實驗結果，這個系統在正常的運作情況下需要花費少量的時間來記錄狀態。錯誤回復的速度也在可以接受的範圍。

The Design and Implementation of Fault Tolerant Proxy Server

Student：Chia-Chih Wu Advisor：Prof. Ruei-Chuan Chang

Institute of Computer and Information Science

National Chiao-Tung University

Abstract

Proxy caching is a widely deployed technique to improve user-perceived latency and usage efficiency of network resources. However, researches show that many reasons may cause the outage of a proxy server. Thus, the clients cannot request web sites through the proxy before the administrator fixes the failure. Although many techniques provide fault tolerance for web caching, the requests which are processed when the proxy fails cannot be recovered.

We purpose a fault tolerant proxy system which can log the state, detect the fault, and recover the on-line requests in a way transparent to clients and servers. We adopt FT-TCP [2] techniques to recover the connections. The system is implemented in the kernel-level. TCP traffic and system calls are intercepted to perform state logging and recovery. According to the experimental results, the system has low overhead and acceptable performance.

Acknowledgments

For accomplishing this thesis, I deeply appreciate the guidance of my advisor, Prof. Ruei-Chuan Chang, for instructing me in research and for providing the abundant resources for study and experiments. Additionally, I deeply appreciated Dr. Da-Wei Chang's assistance for giving me much advice and opinions on revising this thesis. Thanks to each member of the computer system laboratory for their encouragement and kindly help.

I'd like to thank my parent, too, for their encouragement and unlimited love. Finally, I want to thank all my friends for all the joyous things that inspire my life.

Chia-Chih Wu

Institute of Computer and Information Science

National Chiao-Tung University

2004/6

Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

World Wide Web (WWW) has become the most popular application on the Internet. Among the WWW technologies, proxy caching is a widely deployed one for improving user-perceived latency and usage efficiency of network resources. By caching web pages in a proxy server, many requests can directly be satisfied from the proxy instead of the web servers.

However, proxy systems may fail. According to the previous studies [7] [12] [20], human error is the main source of system failure. To take an example, a proxy application may be killed by the administrator carelessly. Besides, transient faults or the software aging [8] problem may cause a proxy to crash. Such unintentional proxy termination has impacts on the clients who access web sites through the proxy.

Proxy failures often result in performance loss. Moreover, in some cases, clients are forced to rely on proxies to access web sites on Internet and therefore the failure of the proxies leads to service unavailability. For example, some services (e.g., e-magazines) only accept requests from registered IP addresses in order to restrain the access from unregistered clients. For saving the subscription cost, an organization usually uses the IP address of the proxy to subscribe the service. This makes all the staffs in the organization be able to access the service. However, if the proxy fails, the service will become unavailable. For another example, many companies and schools use proxy technique to provide WWW access for a large number of hosts through a few number of public IP addresses. The failure of the proxy will cause the WWW channel to break until the administrator fixes it.

Previous techniques have limitations when they are used for building fault tolerant proxies. Specifically, simple mechanisms based on proxy replicas [6, 9, 10, 14, 19] can not recover the requests that are served when a proxy service fails (i.e., on-line requests). Connection migration techniques [4, 22, 23, 24, 28] can recover on-line requests. However, they can not do it in a server transparent way. They *replay* the requests from the beginning again, which makes the proxy to establish connections to the server and issue the requests one more time. The replay not only wastes the sever resources and increases the recovery time, but also cause problems for dynamic-object or transaction-based requests. Some connection migration techniques [22, 23, 24] even require modifications to the client-side TCP implementations, which limits the feasibility of these techniques on providing fault tolerant proxy services.

The main reason that the connection migration techniques have limitations on providing fault tolerant proxy service is as follows. In their systems, a host only plays a single role, either a client or a server. Therefore, they did not consider how to transparently recover a system that acts as both clients and servers, like proxy.

This kind of connection reestablishment wastes the server resources, increases the replaying time, and may cause problems for dynamic-object requests.

In this thesis, we propose a system to make a proxy service become fault tolerant. It can automatically detect the proxy service failures, restart the proxy, and recover the state (including the on-line requests) associated with the proxy. Thus, the interactions between the clients and the servers would not be interrupted because of the failures. In our system, the TCP connection recovery is based on the FT-TCP [4]. However, unlike the FT-TCP, the recovery is transparent to not only the clients but also the servers. In addition, it is also transparent to both the proxy application and the TCP/IP stack of the proxy.

We implement the fault-tolerant system as a Linux kernel module. The target application is Squid [25], the most popular proxy application in UNIX-like systems. In the original implementation, the overhead of states logging is around 4% and 839 ms is required to recovery 30 client sessions on the proxy server. To reduce the recovery time, we further proposed two optimization techniques. Furthermore, small space is required to store the states.

The rest of this thesis is organized as follows. Chapter 2 describes the background technique, FT-TCP, which is adopted in our system. Chapter 3 surveys various existing solutions to provide a fault-tolerant proxy server. The design and implementation of our system is described in Chapter 4. We show the experimental results and the analysis of the results in Chapter 5. We conclude this paper and describe the future works in Chapter 6.

CHAPTER 2

BACKGROUND

In this chapter, we describe the techniques of FT-TCP [4], which were adopted and modified by us for building a fault-tolerant proxy system.

2.1 ARCHITECTURE OF FT-TCP

The architecture of FT-TCP is presented in Figure 2.1. The main idea of FT-TCP is to intercept all the TCP traffic, record the traffic information to a logger machine, and recover the TCP connections in another backup machine during the recovery period. It is worth to mention that the clients are not aware of the connection migration. All the tasks are performed on the server side and transparent to the clients.

The TCP traffic interception is done by two separated wrapping layers. The layer between applications and TCP is called North Side Wrapper (or NSW). It records the interactions between TCP and the applications (e.g., socket read/write operations) to the logger, and recovers the states (i.e., socket stream states) of the applications during the recovery period. Specifically, the NSW records the data read by the application and the return values of each socket write operation to the logger. To recover the socket stream states, the NSW communicates with the logger and feeds the logged data (or, returns the logged return values to the application), while it performs socket read or write operations. In other words, the NSW makes the application replay socket read/write operations to recover the states. The layer between TCP and IP layers is called South Side Wrapper (or SSW). It records each TCP connection to the logger and helps to re-establish the connections during the recovery period. When TCP sends/receives a packet during normal operations, it intercepts the packet and records the information required for re-establishing the connection (e.g., the sequence number) to the logger. When FT-TCP re-establishes a connection, the SSW modifies (sometimes destroys) the packets sent to/from TCP in order to perform the recovery transparent to the application and the clients.

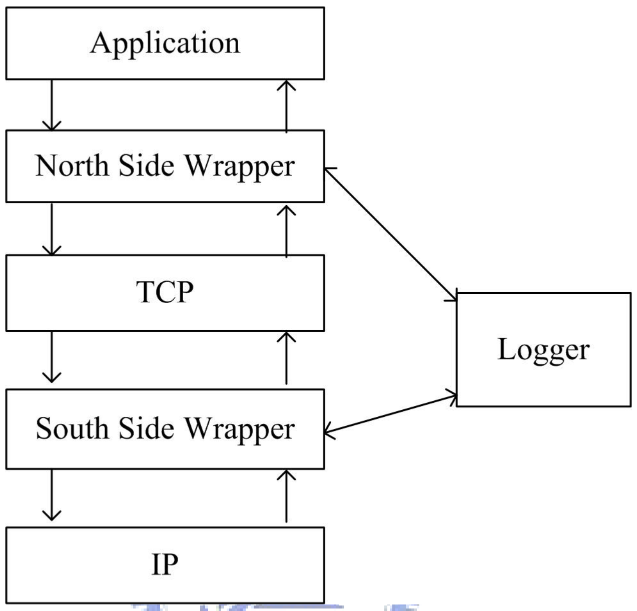

Figure 2.1: FT-TCP Architecture

2.2 CONNECTION RECOVERY

Figure 2.2 illustrates the flow of connection recovery performed in FT-TCP. In order to re-establish a connection with client transparency, FT-TCP spoofs and intercepts packets to complete a fake 3-way handshake with the server-side TCP, which is described in the following.

After the server restarts, FT-TCP spoofs an SYN packet to the server-side TCP with the initial sequence number as the last ACK sequence number sent by the server. So that, the sequence number of the next incoming packet in this new connection will follow the number of the original one. When getting the SYN packet, the server's TCP stack sends a SYN/ACK packet back, which is intercepted by the FT-TCP. The FT-TCP destroys the packet. In addition, it records the *delta_seq*, the difference between the initial sequence numbers of the re-established and the original connections. The *delta_seq* is used for adjusting the sequence numbers of the following TCP packets in order to maintain server-application and client-side transparency. Finally, FT-TCP fakes an ACK packet to complete the 3-way handshake.

After the connection is re-established, the sequence numbers of all packets in this connection should be adjusted. Specifically, FT-TCP adjusts the ACK sequence number of the incoming packets and the sequence number of the outgoing packets. Needless to say, the checksum should be re-computed after the sequence number adjustment.

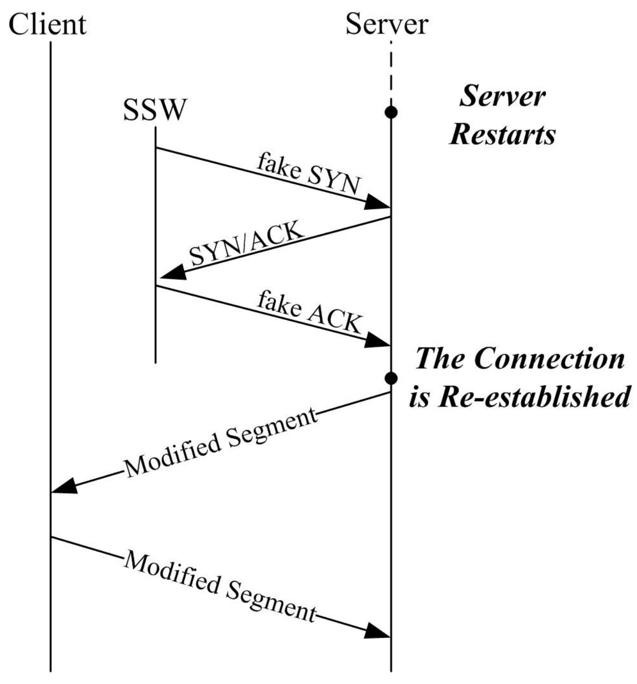

Figure 2.2: The Flow of Connection Recovery in FT-TCP

CHAPTER 3

RELATED WORK

Previous techniques for providing proxy fault tolerance can briefly be classified into two categories, proxy replicas and connection migration. We will describe these techniques in this chapter.

3.1 PROXY REPLICAS

A straightforward approach to provide proxy fault tolerance is to use multiple proxy replicas. Based on this approach, many techniques can be used. Proxy auto-configuration script [19] enables the clients to choose another proxy while the primary one fails. DNS aliasing [6] can be used to select a proxy in a round-robin manner. This reduces the impact of a single proxy failure. Moreover, this technique can also be enhanced to select the live proxies only [9] to eliminate the problem of proxy failure. However, the effectiveness of this technique may be reduced due to the caching of the DNS results on the clients and other DNS servers. Specifically, a client can’t connect to another live proxy before issuing the DNS query again once the cached DNS result targets to a failed proxy. ARP spoofing and IP aliasing [14] techniques can also be used to provide fault tolerance for proxies. When the primary proxy fails, the backup proxy takes over the IP address of the primary one by sending gratuitous ARP packets. Therefore, following requests will be directed to the backup proxy. All the techniques described above are client-transparent except for the proxy auto-configuration script. However, they do not support for recovering the requests that are being served when the proxy fails.

Web Cache Communication Protocol (WCCP) [10] allows a router to communicate with multiple proxies so as to enable transparent caching. Using this protocol, a layer-4 router is responsible for intercepting web traffic and routing the web requests to the proxy hosts. In addition, the router also detects the state of each proxy machine so as to route the requests to well-functioned proxies. The drawback of this approach is that it requires the router support. Moreover, it cannot recover on-line requests.

Squirrel [15] uses a peer-to-peer routing protocol to provide distributed web caching on client desktop machines. Since each client machine cooperates in the peer-to-peer environment, the departure or failure of one Squirrel node could be handled properly. However, similar to the above approaches, it cannot recover on-line requests.

3.2 CONNECTION MIGRATION TECHNIQUES

Connection state migration in our fault-tolerant proxy subsystem is based on FT-TCP [4, 28], which recovers the TCP connection state of a server in a client-transparent way. It logs the connection information and the related server process states, so that the un-closed connections can be re-established and the states can be recovered when the server fails. However, FT-TCP is not suitable for proxy applications. Specifically, FT-TCP will re-establish new proxy-server connections and forward requests to web servers again while recovering a proxy. This causes problems such as inconsistent dynamic content[1] and duplicated transactions for dynamic-object or transaction-based requests. In addition, it consumes an unnecessary long time.

The approach proposed in [23] has the ability to migrate TCP connections across server replicas. When a server fails, one of the replicas will re-establish the client connections which were previously managed by the failed server and resume the service. Although this approach operates transparently to the client application and the server application, the connection

[1] The resulting content of a dynamic page becomes a mixture of two different responses due to the replay. This will be mentioned in Chapter 4.

migration mechanism requires modifications to both the client-side and the server-side TCP implementations. Specifically, the TCP implementations should be extended to support the TCP Migrate Options [22]. For proxy service, it is difficult to deploy the TCP extension to all the hosts interacting with a proxy. Migratory TCP (M-TCP) [24] is a transport layer protocol which enables the resumption of a failed service in the fashion of connection migration. A degradation in quality of service triggers the migration, which makes the client to reconnect to a better performing server replica. A set of API is provided for the server applications to support state transfer between server replicas. Similar to the approach used in [23], the TCP stacks on both the client and the server require modifications to achieve connection migration. On the other hand, our approach does not require any modifications to the client-side and server-side TCP implementations. Thus, it is much easier to be deployed.

In addition to the research efforts in the above two categories, there are also projects that focus on providing fault-tolerance web systems [1, 2, 26, 27]. These systems log the TCP/IP and HTTP information in order to achieve seamless service failover. Moreover, they eliminate the problem of inconsistent dynamic result by delaying the sending of a HTTP response to the client until the response is fully generated. However, they still cannot achieve server transparency while the mechanisms are applied on proxies. This is because the on-line requests will be replayed (by the backup proxy) from the beginning again. For transaction-based requests, the replay will cause the transaction to be executed again. This may lead to problems in the server or make the user be charged twice.

CHAPTER 4

DESIGN AND IMPLEMENTATION

In a normal HTTP system with a proxy server (as shown in Figure 4.1), a client sends a HTTP request to the proxy, which reads the cached web page from its memory (or disk) or forwards the request to the web server. And then, the proxy sends the requested page to the client. In the meantime, the proxy may crash or be unintentionally terminated by the administrator, making the underlying TCP/IP protocol stack send out FIN packets for all the connections belonging to this process. As a result, all the connection states will be lost. In order to avoid this problem, we have to modify the existing TCP/IP protocol implementation to transparently re-establish the connections after restarting the proxy.

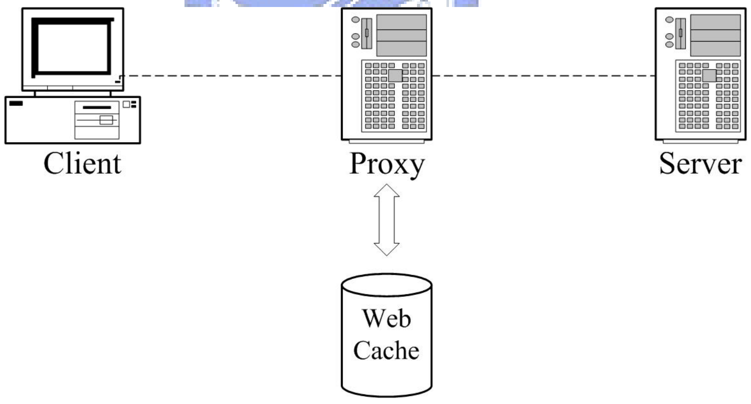

Figure 4.1: Simple HTTP Proxy Architecture

We adopt the techniques used in FT-TCP [4], which provide a recoverable TCP/IP protocol stack suitable for web server and other network services [28]. We have described the techniques in Chapter 2. However, using FT-TCP for proxy recovery faces two problems.

First, FT-TCP cannot recover the proxy-server connections. Since FT-TCP replays all operations while performing recovery, the proxy will reconnect to the server and forward the

HTTP request again after it restarts. This lacking of server transparency may cause problems for transaction-based or dynamic object requests. In the former case, the server will start two transactions while the client issues a request only. In the latter case, two requests may result in two different response objects, and the object returned to the client may be a mix of the both response objects, which is obviously an incorrect result.

Second, FT-TCP consumes an unnecessary long time, which can be reduced in our work, while recovering a proxy server. This is due to the communication with the long-distance servers while performing replay. In our work, the replay is performed locally (without interacting with the servers) which reduces the communication time.

In the following of this chapter, we will describe the design and implementation of our fault-tolerant proxy system. Section 4.1 describes how to record the information that is required for recovery. The flow of recovering a proxy server is presented in Section 4.2. Section 4.3 describes how to re-establish the connections. Section 4.4 describes the way to recover the data in socket streams. The recovery approach of file stream data will be presented in Section 4.5. Finally, we will show the detailed implementation of our system in Section 4.6.

4.1 SESSION-BASED LOGGING

In order to recover the state of a proxy application after it restarts, we have to record the states during its normal operation period. Instead of taking snapshots of the whole process state, we record what the proxy did for each client and recover the on-line client sessions only.

A client session starts when a client request arrives to the proxy. The proxy may serve the request directly from its cache or get the requested page from the server. After the proxy returns the page completely to the client, the session terminates. Therefore, the states in a client session include the state of the client-side and the server-side socket streams, and the state of the cache file streams. Although the state is available in the kernel, it is still a challenging problem to associate the states with the client sessions. Association of the client-proxy connections can be achieved through the use of the (client IP address, client TCP port) pair since the pair is unique for each client session. However, the other information (i.e., opened file and proxy-server connection) can't easily be associated to a client session without modifying the proxy application.

In Section 4.1.1, we will describe how to associate the required states to the corresponding client session. In Section 4.1.2, we will show the way to log the streams states which is needed while recovering a proxy. The cases of information logging will be presented in Section 4.1.3.

4.1.1 Stream-Session Association

As we described above, we can associate a client-proxy connection to the corresponding client session. However, associating the proxy-server connection and the opened cache file to the session is not easy. In the following, we will describe how to do this in a way transparent to the proxy application.

The first step is to establish the mapping from a proxy-server connection to the corresponding client session. Since two different clients may request the same web site, even the same URL, we can't differentiate these connections by the client IP addresses, TCP port numbers, or even the client requests. One way to workaround this problem is to allow the proxy application to give a hint about the client-proxy connection when it establishes a proxy-server connection. However, this approach requires modification to the proxy application.

To avoid modifying the proxy application, we take another approach. We use the "Pragma" field in HTTP header to carry the session identifier. Figure 4.2 illustrates the flow of the mapping-establishment. When a HTTP request arrives (step 1), the NSW hijacks the HTTP request, parses it, and appends a string “pragma: sid=*n*” into the HTTP header, where *n* is the session identifier assigned by the SSW when the 3-way handshake completes. Then, the request is handed to the proxy application (step 2). According to HTTP standard [13], the “Pragma” directives must be passed through by a proxy or gateway application. Therefore, the proxy forwards the modified request to the server (step 3). The request is again intercepted by the NSW, which then extracts the session identifier to establish the mapping of the proxy-server connection to the client session. Finally, the request is sent out to the server (step 4).

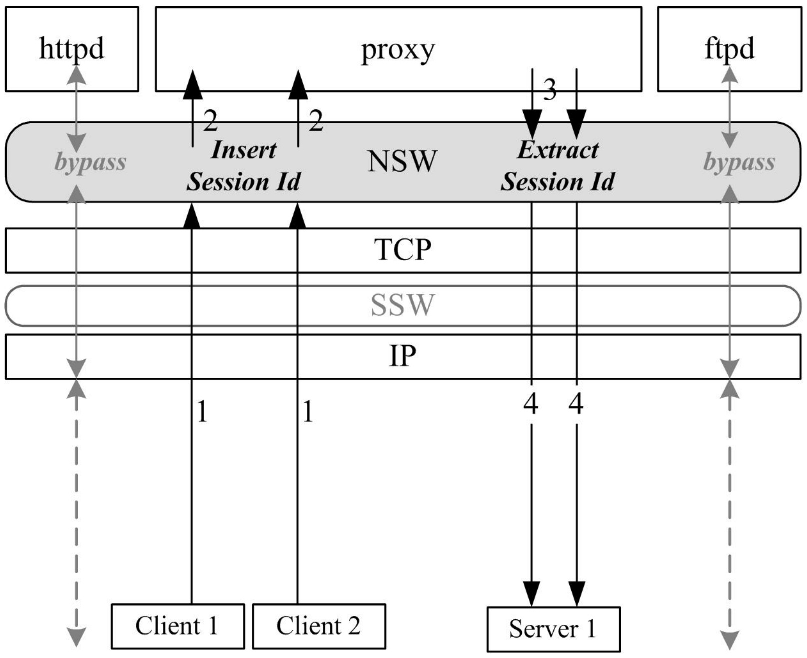

Figure 4.2: Relating the Proxy-Server Connection with the Session

The overheads of this approach are the monitoring of socket read/write operations and the insertion/extracting of the session identifiers. According to the experimental results, these overheads have little impact on the performance. Moreover, the overheads only occur on the proxy applications. As show in Figure 4.2, the overheads don’t occur in the httpd and ftpd paths.

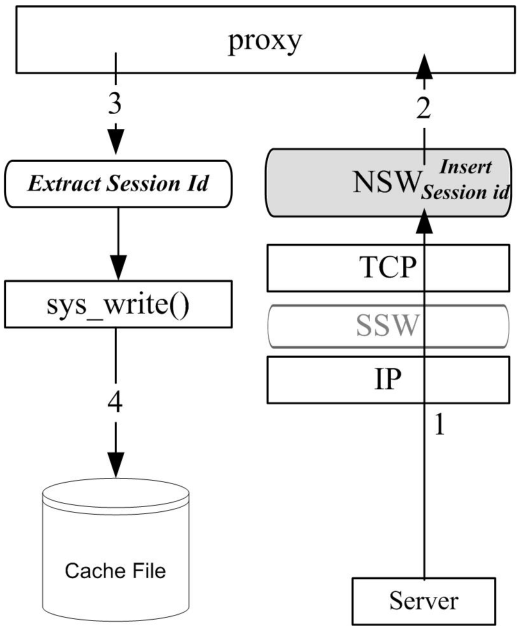

Figure 4.3: Relating the Cache File with the Session

In the following, we describe how to relate the cache file with the client session. Since Squid writes a full HTTP response including the HTTP header to the cache file, we can apply a similar solution as we proposed above. Figure 4.3 illustrates the flow of establishing the mapping between the cache file name and the client session. When a HTTP response arrives (step 1), the NSW inserts the client session identifier into the response header and hands the modified response to the proxy application (step 2). If the proxy application writes the response to a cache file (step 3), the write() system call will be intercepted and the session identifier will be extracted to establish the mapping. After the mapping has been established, the response is written to cache file via the original write operation (step 4).

4.1.2 LOGGING OF STREAM STATES

After describing how to associate streams with the client sessions, we present the way to record the states of each stream belonging to the proxy application. In a proxy, the socket streams can be divided into two parts, client-side and server-side streams. In addition to the socket streams, the states of the file streams are also required to recover. For the three kinds of streams, the states needed to be logged are different due to their availability when the proxy restarts.

Basically, the data transmitted within these streams are HTTP requests and responses. For example, a proxy receives HTTP requests from the client streams, receives HTTP response from the server streams, and writes HTTP response (including the header) via the file streams. Each on-line HTTP request should fully be recorded in order to re-issue it during the recovery period. However, we don't keep the full HTTP responses. Only the header and the unhandled data[2] of a response are stored in the log buffer. The former is kept in order to make the proxy application normally processes the HTTP response after it restarts. The latter is logged because it is no longer available from other places.

Figure 4.4 illustrates the unhandled data logging. The number W, P, and Q are the numbers of bytes received from the server, written to the cache file, and sent to the client, respectively. Since the proxy receives data before processing it, the value of W is larger than or equal to the maximum value of P and Q. In this figure, we store the data between P and W. The bytes from Q to W are kept since they are not transmitted to the client yet. Moreover, data bytes from P to Q also need to be stored since they reflect the data that was sent to the client but not yet stored into the cache file. In the case that Q is smaller than P, we kept the data between Q and W in the log buffer. In summary, data bytes between W and the minimum number of P and Q are stored in the log for recovery purpose.

[2] Unhandled Data stands for the data that is received within each stream but not yet been processed (i.e. stored into the cache file or sent to the client).

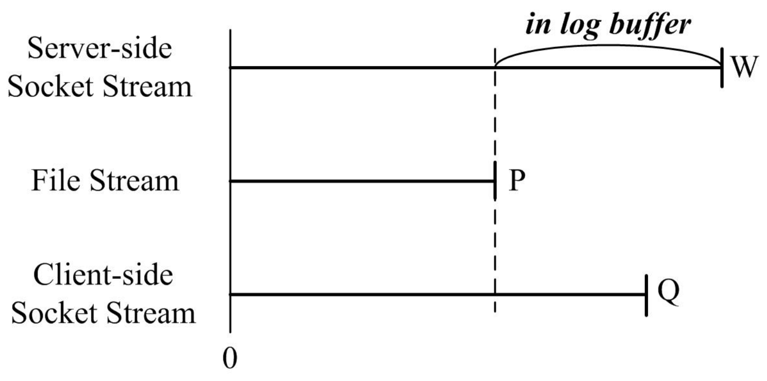

Figure 4.4: Logging of a HTTP Response

It is worth to mention that, in Figure 4.4, *W* and *Q* are implemented based on the sequence numbers in TCP, while the implementation of *P* is based on the file offset.

4.1.3 CASES OF INFORMATION LOGGING

The control flow of a proxy varies according to whether the object requested by the client is in the cache or not, and the cacheablility of the object. We divide the flows into three cases and describe the information that need to be recorded for each case.

The first case is that the requested object is in the cache. The proxy reads the object, forms the HTTP response, and sends the response back to the client. In this case, we should record the information of the client-proxy connection so that we can reestablish the connection after restarting the proxy. In addition, we also have to record the stream state of this connection so that the proxy can continue sending/receiving data with the client after it restarts.

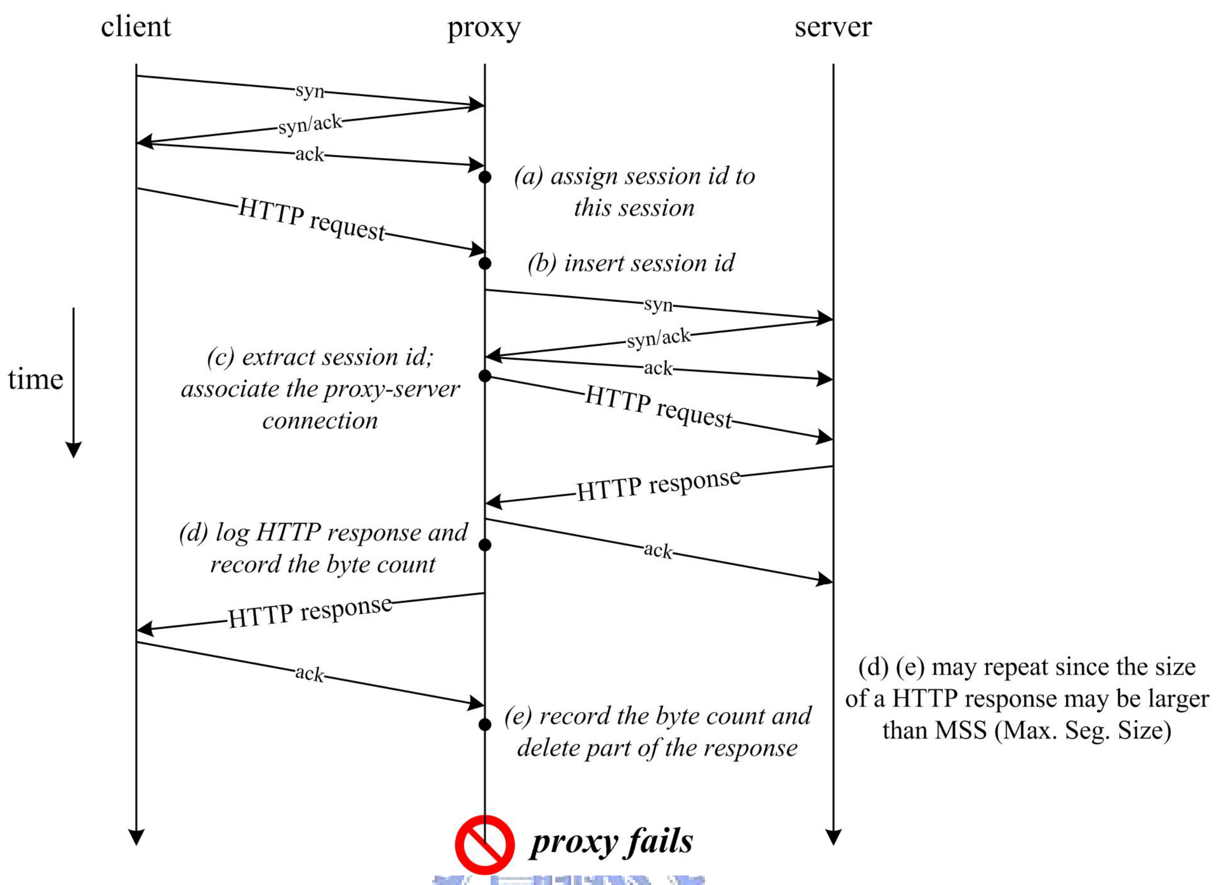
client
proxy
server
syn
syn/ack
ack
(a) assign session id to this session
HTTP request
(b) insert session id
syn
syn/ack
ack
time
(c) extract session id; associate the proxy-server connection
HTTP request
HTTP response
ack
(d) log HTTP response and record the byte count
HTTP response
ack
(e) record the byte count and delete part of the response
(d) (e) may repeat since the size of a HTTP response may be larger than MSS (Max. Seg. Size)
proxy fails

Figure 4.5: the second case of information recording

The second case is that the requested object is not in the proxy's cache and the object is not cacheable. The proxy gets the requested page from the web server and sends it back to the client. However, after examining the HTTP header, page size and other information, the proxy decides not to cache this page. In addition to logging the information mentioned in the previous case, we have to record the information of the proxy-server connection. Figure 4.5 illustrates the flow of information logging in this case. When a HTTP request arrives at the proxy, we record the complete request and insert the session identifier (that is assigned while establishing the client-proxy connection (a)) into the request (b). After establishing the proxy-server connection, the proxy forwards the request to the server. We parse the request to extract the session identifier and associate the proxy-server connection (specifically, the source port of the connection) with the session (c). Each time a HTTP response packet arrives, we append it to the log buffer and update the byte count accordingly (d). Similarly, we record the number of bytes received by the client and delete part of the response from log buffer accordingly (e).

The last case is similar to the second one except that the proxy decides to cache the response in the storage. Besides logging the information in the second case, we also record the states of file streams. We don’t mention the details of logging a file stream. It is similar to the approach of logging a socket stream, which was described in the previous paragraph.

4.2 THE FLOW OF RECOVERING

Figure 4.6 illustrates the flow of recovering a session. When the proxy restarts, the SSW first re-establishes the client-proxy connection (a). The HTTP request will be resent to the Squid after the connection re-establishment (b). This way make the proxy connect to the server and forward the request. Similarly, the server connection re-establishment is performed by the SSW (c). After both connections are re-established, the HTTP response is fed to the proxy application to recover the socket stream states and the cache file stream states (d). Note that part of the HTTP response was already received by the client, thus we drop them during the recovery. Finally, the HTTP traffic could be transmitted normally.

In the following sections, we will describe the details of the recovery flow.

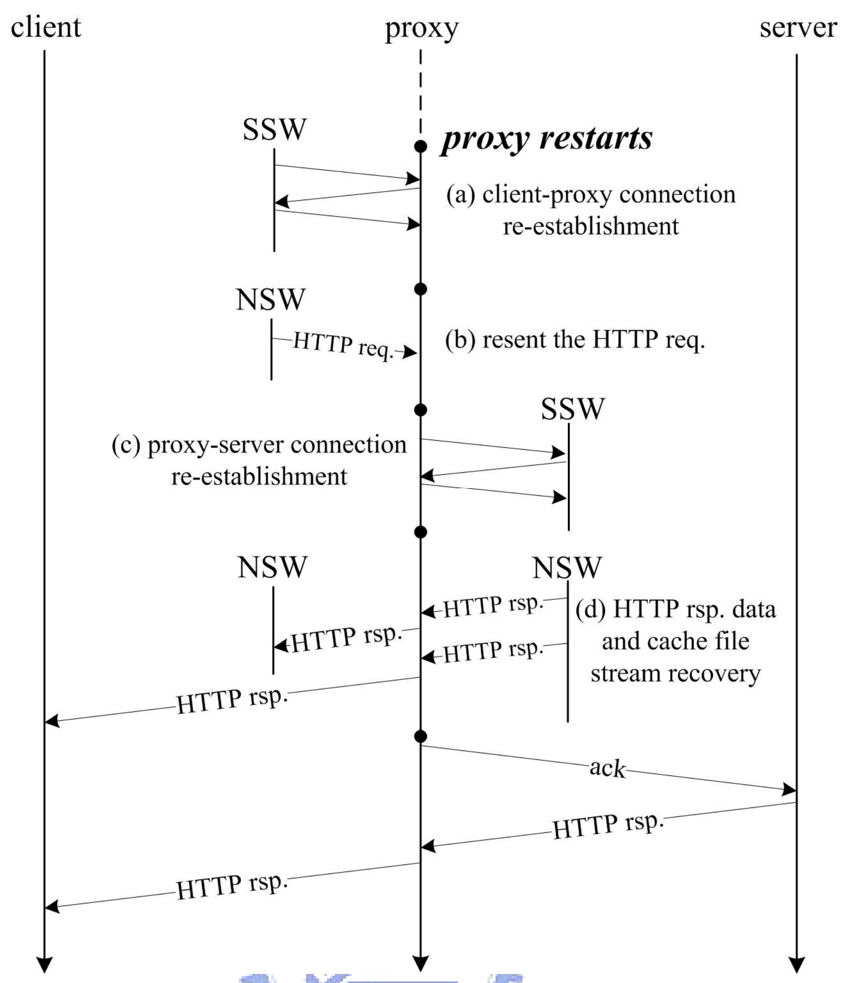
client
proxy
server
SSW
proxy restarts
(a) client-proxy connection re-establishment
NSW
HTTP req.
(b) resent the HTTP req.
SSW
(c) proxy-server connection re-establishment
NSW
NSW
HTTP rsp.
HTTP rsp.
HTTP rsp.
(d) HTTP rsp. data and cache file stream recovery
HTTP rsp.
ack
HTTP rsp.
HTTP rsp.

Figure 4.6: The Flow of Recovery a Session

4.3 Connection Reestablishment

In this section, we show how the connections are re-established according to the logged information. For proxy-client connections, we adopt the same techniques proposed in FT-TCP [4], which was described in Chapter 2. For proxy-server connections, which cannot transparently be re-established in FT-TCP, we modify the FT-TCP techniques to perform the connection re-establishment in a way that transparent to the proxy application.

While re-establishing a client-proxy connection, the SSW fakes SYN packet to the TCP stack to make it feel that it is accepting a new connection. After receiving the HTTP request (which is also spoofed by the proxy’s kernel), the proxy application starts connecting to the server. This makes the underlying TCP stack send out a initial SYN packet, which is discarded by the SSW. After discarding the SYN packet, the SSW fakes a SYN/ACK packet and sends it to the TCP layer. However, during the recovery period, new client requests (i.e., new sessions) may arrive and thus making the proxy application build new connections to the servers. The SYN packets for these server connections should not be discarded. Instead, they should be sent to the corresponding servers. Therefore, the SSW should determine whether a SYN packet corresponds to a failed connection or a new one.

We achieve this by controlling the incoming of the HTTP requests. Before all of the failed server connections are re-established, we process one HTTP request at a time. Specifically, when an incoming HTTP request arrives at the NSW, it blocks further requests until the proxy sends an outgoing SYN packet to the server or returns a HTTP response back to the client. Therefore, when the proxy sends an outgoing SYN packet, we can tell whether or not the connection is for a new session by checking the (client IP, client port number) pair. If the pair matches one of the existing sessions, the connection corresponds to the failed session and the SYN packet should be dropped. Otherwise, it corresponds to a new session, and the SYN packet should be sent to the server.

In addition to the problem mentioned above, we have to modify each packet header after the proxy restarts so as to continue communicating with the clients and the servers. Specifically, we may adjust the sequence number, acknowledgement sequence number, and destination port. Figure 4.7 illustrates the packet header adjustment. The numbers in the brackets indicate the (sequence number, acknowledgement sequence number, source port, destination port). Assume that the proxy originally connects to the server via port u, and reconnects to the server via port v. Furthermore, the difference between the initial sequence numbers of the original and the re-established connections is δ. For each outgoing packet, the SSW adjusts the sequence number p to $(p+\delta)$ and the source port v to u. For each incoming packet, the SSW adjusts the acknowledgement sequence number *s* to *(s-δ)* and the destination port *u* to *v*.

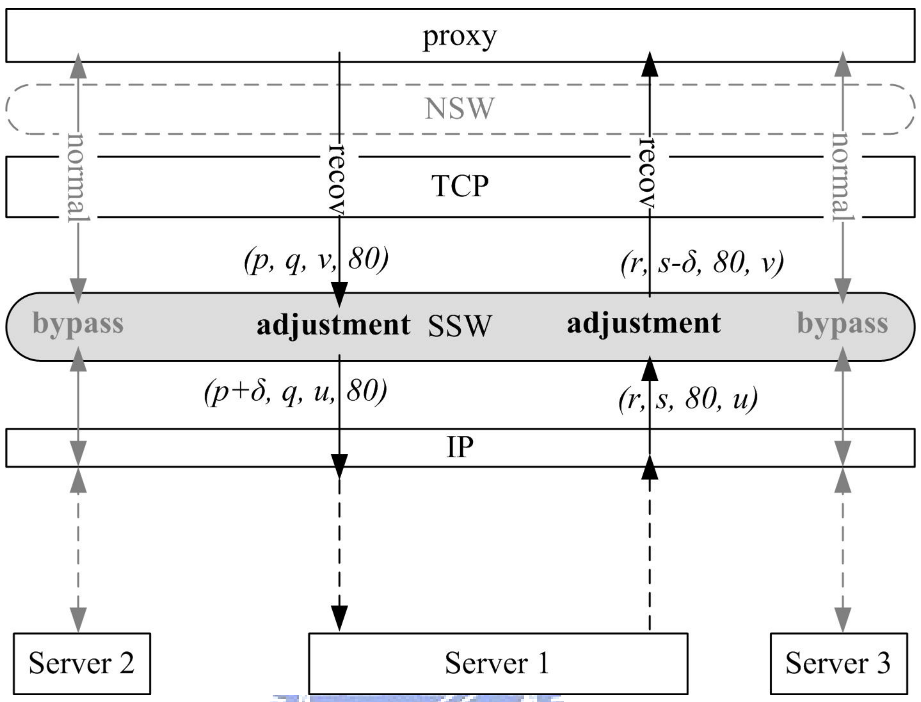

Figure 4.7: Packet Header Adjustment after the Proxy Restarts

4.4 SOCKET IO STREAM RECOVERY

After a proxy restarts, we have to recover the states of the socket streams so that we can continue transmitting/receiving data through the connections. For the client-side socket streams, we adopt the same approach as FT-TCP, which was described in Section 2.2. In this section, we will describe the approach of recovering the states of server-side socket streams.

After the proxy re-establishes a connection to the server, it forwards the logged HTTP request to the server. However, to maintain the server transparency, we cannot actually send the request to the server. Instead, the NSW accepts the request and triggers the recovery of the HTTP response. From the recovery point of view, a HTTP response can be divided into three parts, as shown in Figure 4.8. There are two stages while performing the recovery, which will be described in the following.

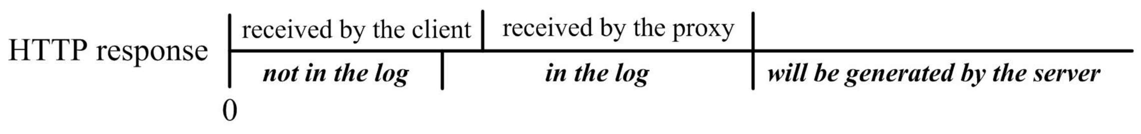

Figure 4.8: HTTP response decomposition

The first stage is to recover the receiving states for the data that is already received by the client. Since the data has been received by the client, we should not send it to the client again. However, for proxy application transparency, the data should be sent to the restarted proxy application. From Figure 4.8 we can see that, only a part of the data is in the log buffer. For the data bytes that are logged, we can send it to the proxy application. For the other data bytes, we generate dummy data and send it to the proxy application. This works because a proxy application doesn't watch the content of a HTTP response. The only exception to this is the response header. A proxy will examine the header and decide whether or not cache the page. Therefore, the logged HTTP response header is returned when the proxy application performs the socket read operation at the first time.

The second stage is to recover the data which is already received by the proxy but not yet received by the client. We can find the data in the unhandled HTTP response buffer (as we described in Section 4.1.2). The NSW intercepts the socket read operation and returns the corresponding data from the buffer. And then, the proxy can forward them to the client and decide whether or not to store the data into the cache file.

After the two-stage recovery, the proxy can receive the remaining HTTP response from the server through the re-established connection.

4.5 File IO Stream Recovery

If our target application, Squid, crashes before storing a full HTTP response to the cache file, the file will become incomplete. When the proxy performs recovery, it may open a new cache file and write the received HTTP response into it. As we described in Section 4.4, only a part of the response contains correct data (since it is in the log buffer), and the other part consist of dummy data. As a result, the proxy will create a cache file with incorrect content, which will be sent to a client when the client requests it.

To solve this problem, we must replace the dummy data with the original HTTP response data stored in the incomplete cache file. The steps are as follows. First, we backup all of the incomplete cache files that correspond to the failed sessions in case the restarted proxy application overwrites the files. When the proxy recovers a session, it will open a new cache file and write HTTP response to that file. Therefore, we intercept the *write* system call and replace the dummy data in the source buffer with the from the backup file. After all the dummy data is replaced, the remaining data bytes written by the proxy application will be stored in the new cache file without modification since they are the original response data bytes.

4.6 Implementation

The system is implemented as a kernel module in Linux 2.4.18. Similar to FT-TCP, we insert the NSW and SSW layers by intercepting several socket-related and TCP-related functions. We also intercept write system call to recover the states of file streams. In addition, we add two kinds of kernel threads for fault detection and packet spoofing.

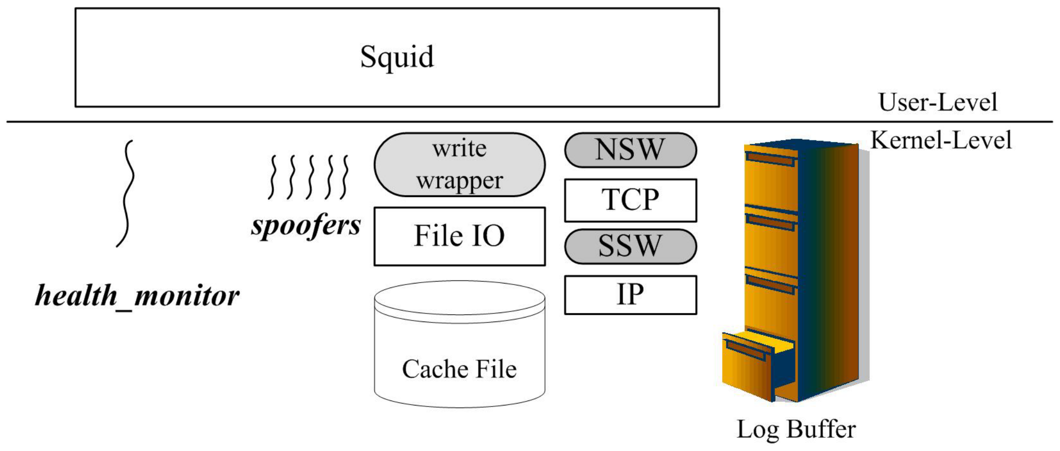

Figure 4.9: Kernel Threads for Fault Detection and Packet Spoofing

For the socket-related functions, we intercept *sock_recvmsg()* to insert session identifier and record HTTP requests. We also intercept *sock_sendmsg()* for session identifier extraction. Besides, these two functions are intercepted to perform the recovery of the socket streams. For the TCP-related functions, we intercept *tcp_v4_rcv()* to hijack the incoming TCP packets. Outgoing TCP packets are hijacked by intercepting the *ip_queue_xmit()* and the *ip_build_and_send_pkt()* functions. All these functions are intercepted in order to record the TCP traffic and establish the fake 3-way handshakes during the recovery period. Finally, for recovering the states of the file streams, we intercept *sys_write()*, which is the in-kernel service routine of *write* system call.

The kernel thread *health_monitor* is used to monitor the status of Squid. It triggers the recovery operations when Squid fails. For each session that needs to be recovered, the *health_monitor* creates a *spoofer* thread, which spoofs packets for re-establishing the corresponding connections (as shown in Figure 4.9).

CHAPTER 5

EXPERIMENTAL RESULTS

This chapter evaluates the fault-tolerant proxy system in four ways. First, we present traces of client-proxy TCP connections to demonstrate the effectiveness of recovering client sessions. Second, we show the overhead on the proxy application and other applications running on the modified operating system. Third, we measure the connection performance of the client sessions which are recovered since the intentional termination of the proxy. Finally, we show the relation between size of log buffer and number of connections and the relation between recovery time and the number of connections.

5.1 EXPERIMENTAL ENVIRONMENT

We use an Intel Pentium 4 2.0 GHz PC running Squid-2.5.STABLE4 [25] as the proxy machine and a server machine which is identical to the proxy machine in hardware running Apache 2.0.40 [5]. Three Pentium 4 1.6 GHz PCs are used as client machines. Each machine is equipped with 512KB cache, 256MB DDR RAM, one 100Mbps Ethernet adaptor, and one Gigabit Ethernet adaptor. All the machines are connected by two independent Ethernet switches (i.e., 100Mbps Ethernet and Gigabit Ethernet).

5.2 Effectiveness

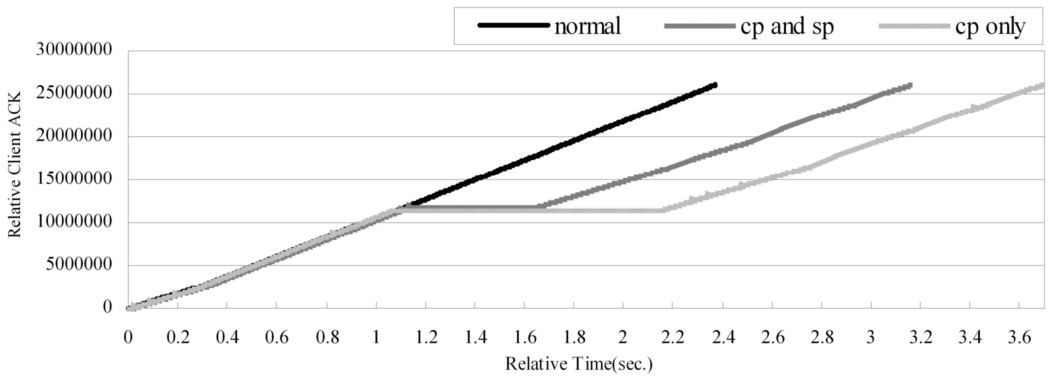

Figure 5.1: Traces of Client-Proxy TCP Connections

In this experiment, the client downloads a 25MB file from the web server through the proxy. We use tcpdump [16] to check the TCP ACK sequence number of each outgoing packet sent by the client. Figure 5.1 shows the results. The *normal* line represents the case that the client completes the downloading normally. In the other two cases, we terminate the proxy application when the client has downloaded 12MB of the file. The *FT-PROXY* line represents the recovery performance of our approach and the *FT-TCP* line represents that of the FT-TCP mechanism. Since FT-TCP make the proxy reconnect to the server and download the file from the beginning again, its recovery time is longer.

5.3 Overhead

In this section, we characterize the overhead of the fault-tolerant proxy system in terms of the impact on the performance of proxy application and other TCP/IP-based applications. In Section 5.2.1, we measure the performance of Squid while logging states of client sessions with WebStone benchmark [17]. We show the overhead occurs in an Apache web server running in the modified operating system in Section 5.2.2.

5.3.1 SQUID PERFORMANCE

To present the impact of states logging on Squid performance, we use the standard workload of WebStone 2.5 but make each request be forwarded to the web server (all request are direct access). We achieve this by modifying the cache policy of Squid. This workload shows the impact of session insertion/extraction to associate the socket streams mentioned in Section 4.1.1.

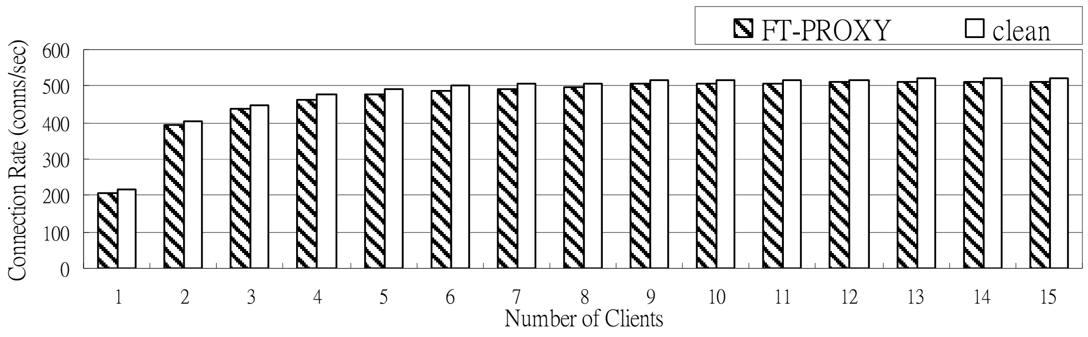

Figure 5.2: Performance of Squid (Connection Rate)

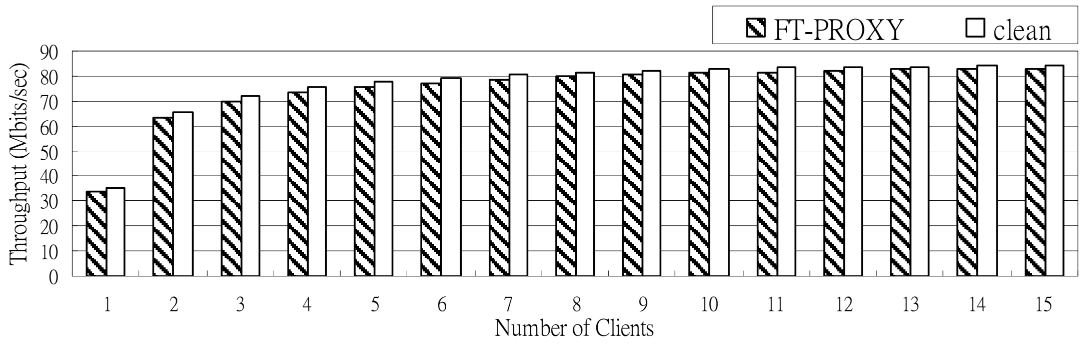

Figure 5.3: Performance of Squid (Connection Throughput)

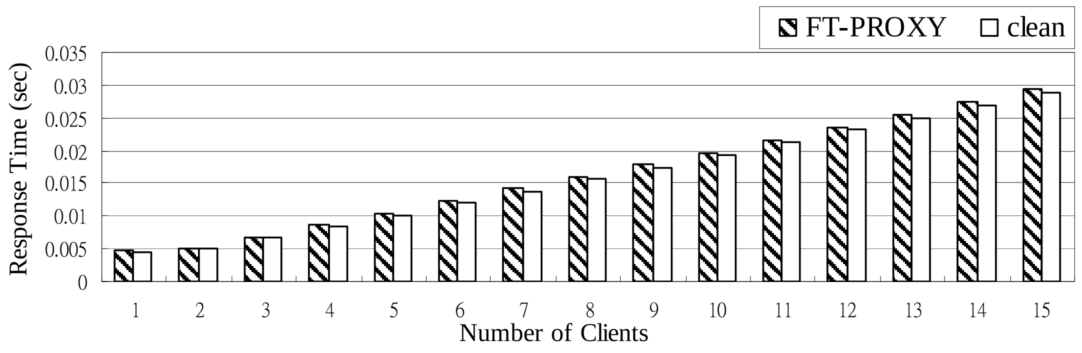

Figure 5.4: Performance of Squid (Response Time)

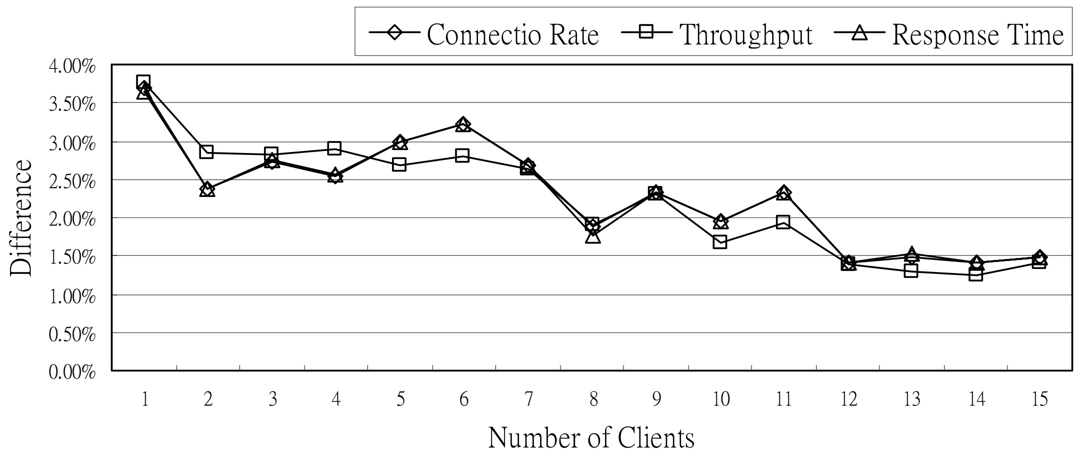

Figure 5.5: Decrease of Squid Performance

Figure 5.2, Figure 5.4, and Figure 5.5 illustrate the runtime overhead caused by the state logging operations. The x-axis stands for the number of clients simulated by WebStone. The shaded bars present the performance of Squid running on the modified operating system (the state logging operations are performed) and the empty bars present the performance of Squid running on the normal operating system. Figure 5.5 shows the decrease of Squid performance in the three criteria. It is worth to mention that the overheads are around 1.5% to 3.8% only.

5.3.2 APACHE PERFORMANCE

We install Apache web server 2.0.40 on the proxy machine. Since the fault-tolerant kernel module checks all TCP traffic and socket read/write operations for logging, the performance of Apache may drop. We measure the overhead with WebStone 2.5. The client machine is the same as Section 5.2.1.

Figure 5.6 illustrates the throughput of the client-server connections. The shaded bars present the performance of Apache running on the modified operating system and the empty bars present the performance of Apache running on the normal operating system. The connection throughput decreases obviously while only one client is simulated. The reason is that the web server has no request to process when the only request is delayed by the modified operating system. Figure 5.7 illustrates the performance decreases of Apache in three ways. Not surprisingly, the performance decrease in each criterion is around 2% while only one client is simulated. The performance decrease of each criterion is lower then 0.5% even lower than 0% while more clients are simulated. Those small or even minus difference numbers are caused by the inaccuracy of measurement. In summary, the overheads of the modified operating system have little impact on Apache web server.

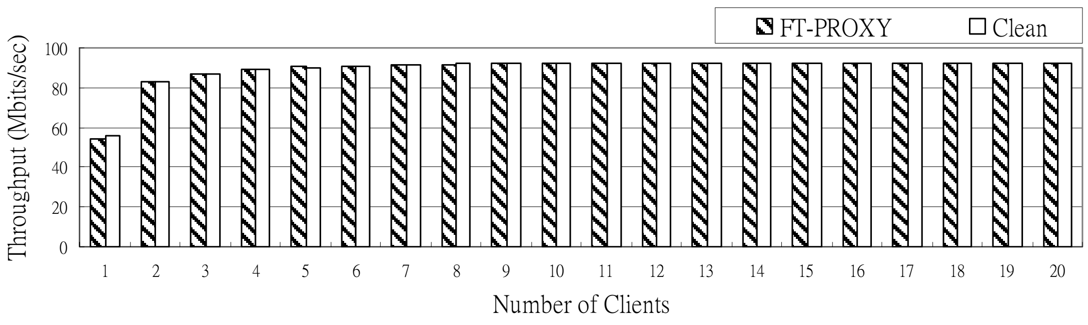

Figure 5.6: Performance of Apache (Connection Throughput)

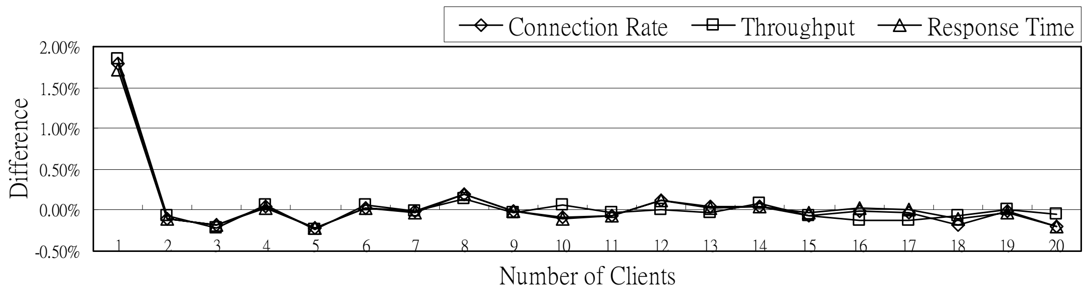

Figure 5.7: Decrease of Apache Performance

5.4 PERFORMANCE

In this section, we measure the performance of Squid which experience failure and recovery. The client requests one file from the server machine through the proxy in each run. We turn off the caching functions of the proxy in order to measure the worst case of recovery latency. Squid process is being terminated intentionally when the client receives 1Kbytes of data in each run. The transmit time is measured in Gigabit Ethernet environment. Table 5.1 shows the results of the experiment. According to the result, the recovery latency is around 720 ms in average.

Since the newly establishing connection (server-proxy connection) cannot receive any packet before completing the re-establishment process, the server-side TCP stack should retransmit the packets which have not been acknowledged. However, the long recovery time of Squid (about 200 ms) leads the expiration of retransmission timer. The server-side TCP retransmits those packets again according to the RTO value, and the value will increase twice if the corresponding ACK packet is not received. By tracking the server-proxy connection, we found that the newly established connection cannot continue transmitting data immediately because of the increasing RTO. We solve this problem by receiving the retransmitted packets in the SSW for the newly establishing connection. According to the result shown in Table 5.2, about 240 ms in average of the recovery latency can be reduced.

File Size (Mbytes)	5	10	15	20	25	30
Normal Transmit Time (sec)	0.258050	0.482425	0.690599	0.925669	1.028636	1.217583
Transmit Time including Recovery Latency (sec)	0.976818	1.228987	1.458313	1.647249	1.768556	1.928813
Recovery Latency (sec)	0.718768	0.746562	0.767713	0.721579	0.739921	0.711230

Table 5.1: Transmit Time of Different Size of File which Experience Recovery

File Size (Mbytes)	5	10	15	20	25	30
Optimized Transmit Time including Recovery Latency (sec)	0.654799	0.961413	1.139041	1.348164	1.478254	1.701996
Optimized Recovery Latency (sec)	0.396749	0.478988	0.448442	0.422494	0.449619	0.484413

Table 5.2: Transmit Time of Different Size of File with Optimization

5.5 SPACE AND TIME REQUIREMENT

Table 5.3 shows the relation between number of on-line sessions and the size of required log buffer. The required buffer size increases while the number of sessions increases. The reason is that we store unhandled data (mentioned in Chapter 4) in the log buffer. Intuitively, the amount of unhandled data increases when the load of Squid increases. Only 132 Kbytes of memory is required to log the state of one session while the proxy is serving 30 sessions.

Number of Sessions	5	10	15	20	25	30
Required Log Buffer (Kbytes)	2.78	28.97	31.06	80.55	127.22	131.94

Table 5.3: Relation between Number of Sessions and Required Log Buffer Size

Table 5.4 shows the relation between number of sessions need to recover and the total recovery time. We intentionally terminate the Squid process when proper number of sessions (e.g., 5, 10, 15) are served. Each client session requests a 25MB web page.

Obviously, to recover more sessions takes more time. And the recovery latency is acceptable.

Number of Sessions	5	10	15	20	30	50
Recovery Time (ms)	102.814	147.626	445.338	525.354	839.004	1751.086

Table 5.4: Relation between Number of Sessions and Recovery Time

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 CONCLUSION

We purpose a subsystem to make a proxy service become fault tolerant. Transient faults on a proxy application (either caused by hardware, administrator or the proxy application itself) can be recovered in a way transparent to the clients, servers, and the proxy application. The subsystem can recover the ongoing requests which are processed when the failure occurs. In addition, the feature of transparency enables the fault tolerant proxy system functions without any support of the clients and the servers. The subsystem is implemented as a Linux kernel module. TCP traffic and system calls are intercepted in kernel-level to log the state. The experimental results show low overhead for state logging and acceptable recovery latency.

6.2 FUTURE WORK

Currently, we focus on the errors happened on a proxy application only. In the future, we will address the operating system failures. Operating system crashes such as kernel panics could be detected by another machine, and the states could also be logged in that machine. Therefore, we can use a logger machine to monitor the proxy. When the proxy fails, the state of the proxy would be migrated to another backup server. Instead of using an additional machine, the system crashes can also be detected by an intellectual network interface card [3]. With the help of fast system restart techniques such as LOBOS [18] which stores the state in a safe memory area that survives after restarting the system, the state recovery can be performed.

In addition, we plan to extend our fault tolerant mechanisms to other network services in the future. In network services such as Web Service [11] and peer-to-peer network [21], each host may play the role of client and server at the same time. This is similar to the proxy service. Therefore, we will evaluate the possibility to extend our approach to provide fault tolerance for those services.

REFERENCES

[1] Navid Aghdaie, Yuval Tamir, “Client-Transparent Fault-Tolerant Web Service,” 20th IEEE International Performance, Computing, and Communications Conference, Phoenix, AZ, pp. 209-216, Apr. 2001.

[2] Navid Aghdaie, Yuval Tamir, “Fast Transparent Failover for Reliable Web Service,” In Proceedings of the International Conference on Parallel and Distributed Computing and Systems, Marina del Rey, California, pp. 757-762, Nov. 2003.

[3] Alacritech, “Alacritech quad port server accelerator,” available at http://www.alacritech.com/html/100x4.html

[4] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, D. Zagorodnov, “Wrapping Server-side TCP to Mask Connection Failures,” In Proceedings of the IEEE INFOCOM, Anchorage, Alaska, pp. 329-337, Apr. 2001.

[5] Apache Software Foundation, “The Apache Web Server,” available at http://www.apache.org/.

[6] T. Briso, “DNS Support for Load Balancing,” IETF RFC 1794, April 1995.

[7] A. Brown, D. A. Patterson, “To Err is Human,” In Proceedings of the 2001 Workshop on Evaluating and Architecting System dependabilitY, Göteborg, Sweden, July 2001.

[8] V. Castelli, R. E. Harper, P. Heidelberger, S. W. Hunter, K. S. Trivedi, K. Vaidyanathan, W. P. Zeggert, “Proactive Management of Software Aging,” IBM JRD, Vol. 45, No. 2, Mar. 2001.

[9] Cisco Systems Inc., “Cisco DistributedDirector,” available at http://www.cisco.com/univercd/cc/td/doc/product/iaabu/distrdir/dd2501/ovr.htm

[10] Cisco Systems Inc., “Web Cache Communication Protocol,” available at http://www.cisco.com/en/US/tech/tk122/tk717/tech_protocol_family_home.html

[11] Harvey M. Deitel, “Web Services: A Technical Introduction,” Prentice Hall, Aug. 2002.

[12] P. Enriquez, A. Brown, D. A. Patterson, “Lessons from the PSTN for Dependable Computing,” In Proceedings of the 2002 Workshop on Self-Healing, Adaptive and self-MANaged systems (SHAMAN), New York, June 2001.

[13] R. Fielding, J. Gettys, J. Mogul, H.Frystyk, L. Masinter, P. Leach, T. Berners-Lee, “Hypertext Transfer Protocol -- HTTP/1.1,” RFC 2616, Jun. 1999.

[14] S. Horman, “Creating Redundant Linux Servers,” In Proceedings of the 4th Annual Linux Expos, Durham NC, May 1998.

[15] S. Iyer, A. Rowstron, P. Druschel, “Squirrel: A Decentralized, Peer-to-Peer Web Cache,” 21th ACM Symposium on Principles of Distributed Computing (PODC), Monterey, California, Jul. 2002.

[16] Van Jacobson, Craig Leres, Steve McCanne, “tcpdump,” available at http://www.tcpdump.org/.

[17] Mindcraft Inc., “WebStone: the Benchmark for Web Servers,” available at

http://www.mindcraft.com/benchmarks/webstone/.

[18] Ron Minnich, "LOBOS: (Linux OS Boots OS) Booting a Kernel in 32-bit Mode," The Fourth Annual Linux Showcase and Conference, Atlanta GA, Oct. 2000.

[19] Netscape, "Navigator Proxy Auto-Config File Format," available at http://wp.netscape.com/eng/mozilla/2.0/relnotes/demo/proxy-live.html, Mar. 1996.

[20] D. Oppenheimer, D. A. Patterson, "Why do Internet services fail, and what can be done about it?" In Proceedings of the 10th ACM SIGOPS European Workshop, Saint-Emilion, France, Sep. 2002.

[21] Andy Oram, "Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology," O'Reilly, Mar. 2001.

[22] Alex C. Snoeren, Hari Balakrishnan, "An End-to-End Approach to Host Mobility," In Proceedings of the 6th Annual ACM/IEEE International Conference on Mobile Computing and Networking, pp. 155–166, Boston, Massachusetts, Aug. 2000.

[23] Alex C. Snoeren, David G. Andersen, Hari Balakrishnan, "Fine-Grained Failover Using Connection Migration," In Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS '01), Mar. 2001.

[24] K. Srinivasan, "M-TCP: Transport Layer Support for Highly Available Network Services," Technical Report DCS-TR459, Rutgers University, Oct. 2001.

[25] D. Wessels, "Squid Web Proxy Cache," available at http://www. squid-cache.org/.

[26] C. S. Yang, M. Y. Luo, "Realizing Fault Resilience in Web-Server Cluster", In Proceedings of the 2000 ACM/IEEE Conf. on Supercomputing (CDROM), p.21-es, Nov. 2000.

[27] C. S. Yang, M. Y. Luo, "Constructing Zero-Loss Web Services", In Proceedings of IEEE INFOCOM 2001, pp. 1781-1790, Apr. 2001.

[28] Dmitrii Zagorodnov, Keith Marzullo, Lorenzo Alvisi, Thomas C. Bressoud, "Engineering fault-tolerant TCP/IP servers using FT-TCP," In Proceedings of IEEE Intl. Conf. on Dependable Systems and Networks (DSN), pp. 22-26, Apr. 2003.