國立交通大學

資訊科學系

碩 士 論 文

虛擬機器支援無資料遺失之網際網路服務系統錯誤回復與升級

Virtual Machine Support for Zero-Loss Internet Service Recovery and Upgrade

研究生：謝承恩

指導教授：張瑞川　教授

中 華 民 國　九 十 四 年 六 月

虛擬機器支援無資料遺失之網際網路服務系統錯誤回復與升級

Virtual Machine Support for Zero-Loss Internet Service Recovery and Upgrade

研 究 生：謝承恩　　Student：Cheng-En Hsieh

指導教授：張瑞川　　Advisor：Ruei-Chuan Chang

國 立 交 通 大 學
資 訊 科 學系
碩 士 論 文

A Thesis
Submitted to Institute of Computer and Information Science
College of Electrical Engineering and Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer and Information Science

June 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年六月

虛擬機器支援無資料遺失之網際網路服務系統回復與升級

研究生：謝承恩　　　　　　　　　指導教授：張瑞川教授

國立交通大學資訊科學所

論　文　摘　要

近年來網際網路服務越來越盛行，越來越多企業使用網際網路系統提供商業性的服務。根據研究顯示，如果商業性的網際網路系統無法提供服務，將會造成企業重大的損失。然而，有許多因素會使得網際網路無法提供服務，這因素包括系統出錯及系統維護。

因此，我們提出一個無資料遺失的網際網路服務系統，使得網際網路服務能夠永續運作。他能夠自動發現系統錯誤，並且在使用者沒有察覺的情況下，回復所有在錯誤發生時正在服務的請求。並且我們也提供一套方法，能使得網際網路服務系統能在不用關閉服務的情況下做系統維護。我們將在虛擬機器-Xen,及虛擬機器上的作業系統上實作我們的架構。根據實驗結果，這個系統在正常情況只需要花費少量的時間來記錄連線狀態。錯誤回復的速度也在可以接受的範圍。

Virtual Machine Support for Zero-Loss Internet Service Recovery and Upgrade

Student: Cheng-En Hsieh Advisor:Prof. Ruei-Chuan Chang

Institute of Computer and Information Science
National Chiao-Tung University

Abstract

In recent years, Internet services have become more and more popular in our daily life. More and more commercial services are provided through Internet. According to previous research, a few minutes of downtime of a commercial Internet service will lead to a great loss of money for the company. Furthermore, system failures and system maintenance contribute to most of the service unavailability.

In order to keep Internet service running permanently, we provide a framework based on virtual machines to allow zero-loss service recovery and upgrade. The framework can detect system faults automatically and transparently recover the on-line requests. Moreover, it also allows on-line requests to be migrated to an upgraded system. We implemented the framework by modifying an open source virtual machine monitor, Xen, and the Linux kernel on top of Xen. According to the experimental results, the framework causes little overhead and has acceptable recovery time.

Acknowledgements

I deeply appreciate the guidance from my advisor, Professor R. C. Change. He taught me the knowledge of operating system, guided me how to do research, and provided me many resources for accomplishing this thesis. Moreover, I also appreciate Dr. Da-Wei Chang's assistance. He gave me much advice for this thesis and revised this thesis painstakingly.

Besides, I would like to thank my family and my friends for their encouragement and unlimited love. I also thank them for bringing me laugh and knowledge that inspire my life.

Cheng-En Hsieh
Institute of Computer and Information Science
National Chiao-Tung University
2005/6

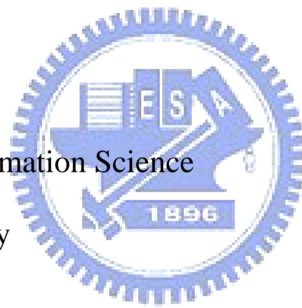


TABLE OF CONTENTS

論 文 摘 要 I
ABSTRACT II
ACKNOWLEDGEMENTS III
TABLE OF CONTENTS IV
LIST OF FIGURES VI
LIST OF TABLES VII
CHAPTER 1 INTRODUCTION 1
CHAPTER 2 BACKGROUND 4
2.1 FT-TCP System 4
2.2 The Recovery Flow 5
CHAPTER 3 RELATED WORK 7
3.1 Fault Tolerance in Application Layer 7
3.1.1 Recursive Restart (RR) 7
3.2 Fault Tolerant in Transport Layer 7
3.2.1 Fine-Grained Failover System 7
3.2.2 M-TCP 8
3.2.3 TCP Splicing 8
3.2.4 FT-TCP 9
3.3 Software Maintenance 10
3.3.1 Devirtualization 10
3.4 Others 10
3.4.1Checkpointing 10
3.4.2 Recovery-Oriented Computing (ROC) 11
3.4.3 Xen 11
CHAPTER 4 DESIGN AND IMPLEMENTATION 12
4.1 System Components 13
4.2 Fault Recovery 14
4.2.1 Backup Server Boot-up 15
4.2.2 Connection State Logging 17
4.2.3 Fault Detection 18
4.2.4 Recovery Flow 20
4.3 Online Maintenance 21
4.3.1 Maintenance Flow 23
CHAPTER 5 RECOVERY TIME REDUCTION 24
5.1 FTP 24
5.2 HTTP Proxy 27

CHAPTER 6 PERFORMANCE EVALUATION 29
6.1 Experimental Environment 29
6.2 Overhead 30
6.2.1 FT-TCP Overhead 30
6.2.2 Squid Performance Overhead 31
6.2.2 Proftpd Performance Overhead 33
6.3 Recovery Time 34
6.3.1 Squid Recovery Time 34
6.3.2 Proftpd Recovery Time 35
CHAPTER 7 CONCLUSION 37
REFERENCES 38

LIST OF FIGURES

Figure 1 FT-TCP....4
Figure 2 Connection Recovery....6
Figure 3 TCP Splicing....8
Figure 4 FT-TCP on Xen....10
Figure 5 Our System Components....13
Figure 6. An Overview of Fault Recovery....14
Figure 7 The Flow of Booting a Backup Server....15
Figure 8 Detecting Application Faults....18
Figure 9 Detect Kernel Fault....19
Figure 10 Recovery Protocol....20
Figure 11 An Overview of the Online Maintenance Flow....22
Figure 12 The Flow of Service Migration....23
Figure 13 A Example of Using FTP for Sending Files....24
Figure 14 Relay Server....27
Figure 15 Performance of FT-TCP....30
Figure 16 Performance of Squid (Connection Rate)....31
Figure 17 Performance of Squid (Connection Throughput)....31
Figure 18 Performance of Squid (Response Time)....32
Figure 19 Performance of Proftpd (Connection Throughput)....33
Figure 20 Fault Occurs When the First 10KB of Data is Sent....34
Figure 21 Fault Occurs When the First Half of Data is Sent....34
Figure 22 Relation between Number of Connections and Recovery Time....35

LIST OF TABLES

Table 1 System Calls Provided by OZS 16
Table 2 Wrapper Registration APIs 25
Table 3 Application-Specific Hook APIs 25
Table 4 Client-Side APIs 26
Table 5 Server Side APIs 28
Table 6 Experimental Environment 29

CHAPTER 1

INTRODUCTION

In recent years, Internet services have become more and more popular in our daily life. However, an Internet service may become unavailable due to transient errors, software bugs, and system maintenance. According to the previous research [19], a few minutes of downtime will lead to a great loss of money. Therefore, high availability is very important for Internet services.

Previous study [20] shows that, software failures lead to a larger portion of system downtime than hardware faults. Moreover, the latter can be masked by component redundancy, such as RAID. Software may crash due to various reasons. As indicated in previous research [2][18], human error is the dominant source. For example, an administrator may misconfigure an Internet service or kill the service unintentionally, which causes the service to become unavailable. Moreover, transient faults or software aging faults [13][15] may occur on the Internet services because of their long execution time. Finally, operating systems under the Internet services can also crash since they are hard to be made error-free due to their high complexity [5]. In addition to software failures, software maintenance is another dominant source of downtime for Internet services [14]. For example, an Internet service has to stop during system maintenance, which may maintain or upgrade the service applications, libraries, operating system, and drivers. Although the system can be restarted after the maintenance operation has completed, the service state and the operating system state (such as the TCP connections) will be lost, which is unacceptable for many commercial or transaction based services.

Many fault tolerant techniques have been proposed to address the problem. However,

they all have limitations on solving the above problem. Checkpointing [22] can not recover software aging faults, and it usually causes a high overhead. Connection migration techniques [1][6][7][23][26] can not recover on-line requests in a server transparent way. Moreover, they require expensive server replicas. Some connection migration techniques [6][7][23] even require modifications to the client-side TCP implementations, which limits the feasibility of the techniques. Recursive Recovery [8][9][10] requires the service to be made up of many fine-grained components, which both needs application re-design and leads to performance degradation. Finally, devirtualization [12] can only deal with planned downtime. It can not cope with unplanned downtime.

In this thesis, we propose a framework to achieve the goal of zero-loss recovery and upgrade for Internet services. Basically, the framework is based on connection migration techniques and virtual machine technology. It can solve the software failure and system maintenance problems mentioned above. Specifically, the framework can log the TCP connection state, detect software faults, and then recover faulty server. Moreover, it requires neither expensive server replicas nor large service program modifications. Some service applications may need to be modified to achieve the goal of zero-loss recovery and upgrade. However, the modification is little and straightforward.

We implemented the framework by enhancing a client-transparent connection recovery technique called FT-TCP[1] and modifying a virtual machine monitor (VMM) called Xen[4] The framework provides the following functionality:

- Fault detection – It can detect faults that occur on service application and the operating system. When a fault is detected, the service recovery procedure will be triggered.
- Connection state logging – The connection state will be logged into the VMM's memory space, and the state will be used to recover the service during the recovery period.
- Recovery management – Once a fault is detected, the framework will recover the service application and the TCP connection state.

According to the experimental results, our approach incurs less than 3.5% throughput overhead. Moreover, it can restart the service with no data loss when the service crashes. Finally, the recovery time is acceptable.

The rest of the thesis is organized as follows. Chapter 2 describes the background technique, FT-TCP. Chapter 3 presents the related work. In Chapter 4 and Chapter 5, we explain our design issues and implementation details. Chapter 6 shows the experimental results, which is followed by the conclusion presented in Chapter 7.

CHAPTER 2

BACKGROUND

As we mentioned in the Introduction, our zero-loss service recovery framework is based on FT-TCP. In this chapter, we introduce the FT-TCP [1] technique and the control flow it uses to recover an Internet service.

2.1 FT-TCP System

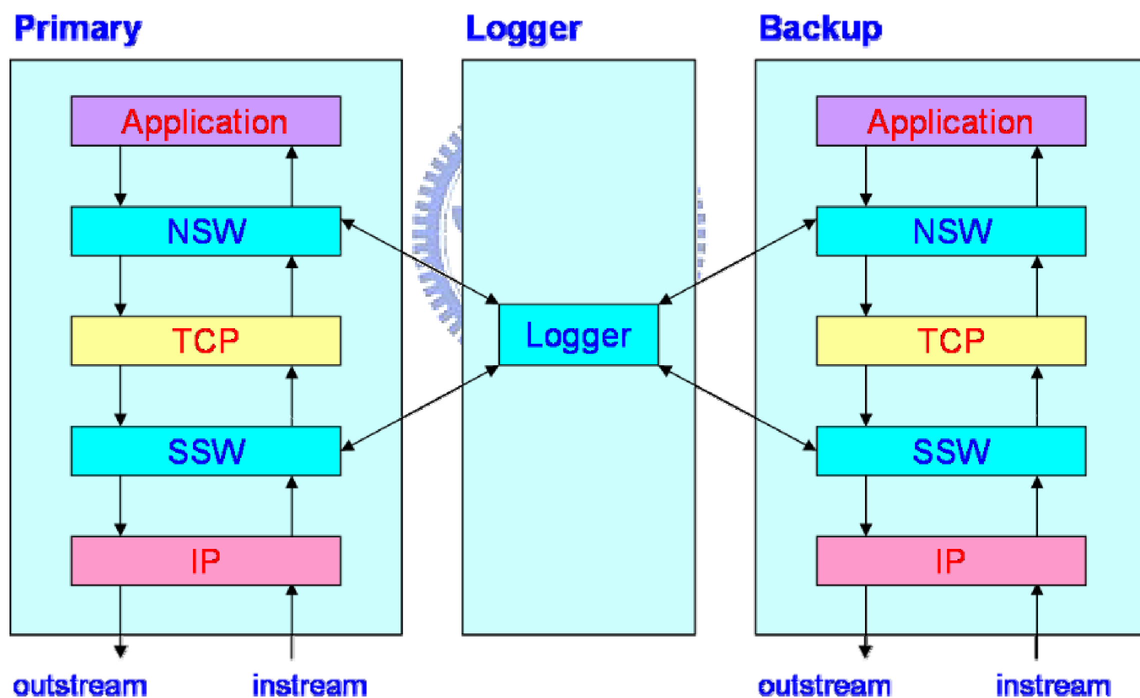


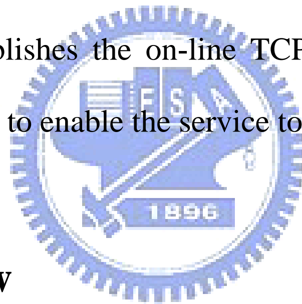
Figure 1 FT-TCP

As shown in Figure 1, FT-TCP inserts wrappers around the TCP layer of the primary machine. The wrappers can intercept all outgoing/incoming TCP packets and record the packet information to the logger machine. When primary machine fails, FT-TCP recovers the on-line TCP connections on the backup machine according to the logged content in an

application-transparent way.

Two wrapper layers are used in FT-TCP. The North Side Wrapper (NSW) sits above TCP. It records socket read/write operations issued by the service application to the logger. During the recovery period, the NSW communicates with the logger to retrieve the logged data, and resends all the pending requests to the application. When the service application issues a socket write operation, NSW is also responsible for returning the same values (which are logged) to the application to maintain the determinism.

The wrapper layer below TCP is called South Side Wrapper (SSW), which 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, SSW intercepts the packet and records the related information such as sequence numbers to the logger. During the recovery period, SSW re-establishes the on-line TCP connections on behalf of the clients according to the logged content to enable the service to continue.



2.2 The Recovery Flow

The flow of connection recovery is presented in Figure 2. In order to recover a connection in a client transparent way, FT-TCP re-establishes the connection on behalf of the client and replays the corresponding socket read/write operations.

For the connection reestablishment, the SSW sends a fake SYN packet to its local TCP (i.e., the TCP of the backup machine). When getting a SYN packet, the TCP sends a SYN/ACK packet back, which is intercepted and then discarded by the SSW. Then, the SSW calculates the delta_seq, which is the difference between the initial server-side sequence numbers of the re-established and the original connections. The delta_seq is used for adjusting the sequence numbers of the following outgoing (i.e., server-to-client) TCP packets and ACK sequence numbers of the following incoming packets in order to maintain client-side

transparency. Note that the sequence numbers of the incoming packets and ACK sequence numbers of the outgoing packets are not needed to be adjusted since SSW use a special initial sequence number, which equals to the last ACK sequence number minus one sent by the primary server, in the SYN packet it fakes. Finally, the SSW spoofs a fake ACK packet to complete the TCP 3-way handshake.

After the connection is re-established, the service application will replay the request-processing flow (i.e, accept the connection, read the request, process the request, and writes the response), which is controlled by the NSW. For example, NSW will send the logged request to the service application when the latter issues socket read operations. For another example, NSW is also responsible for dropping duplicated response data.

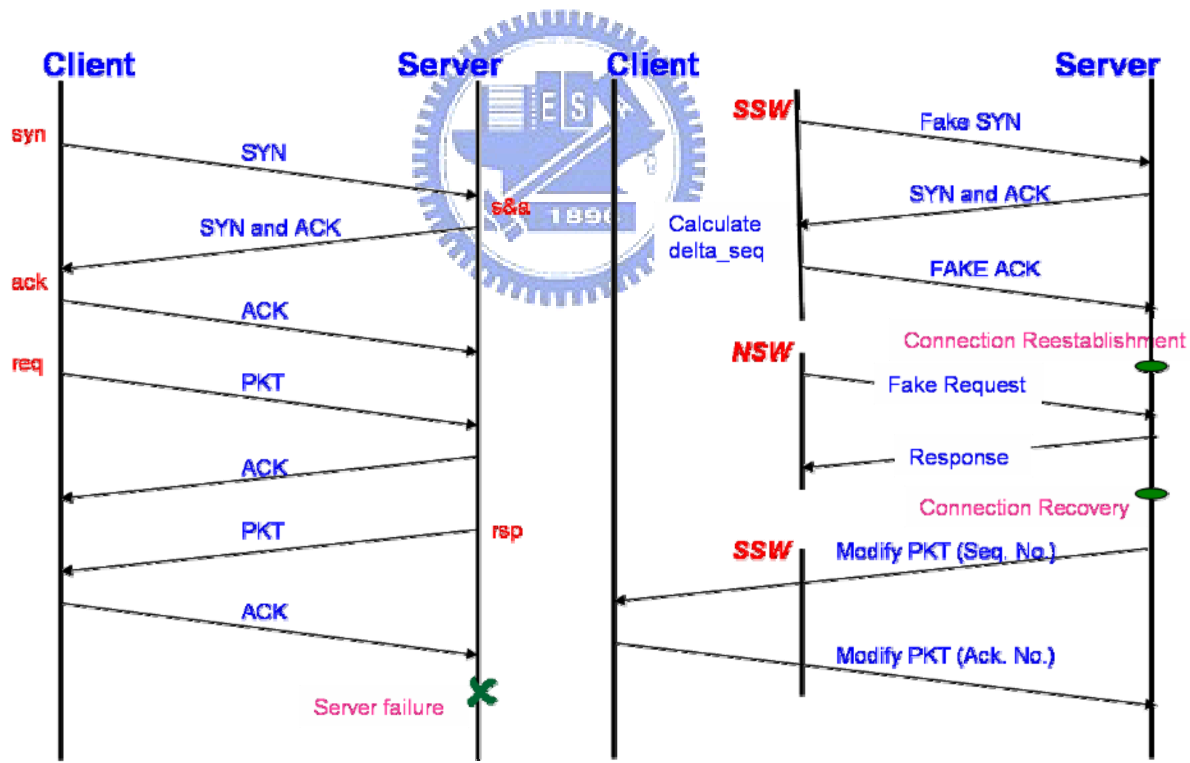


Figure 2 Connection Recovery

CHAPTER 3

RELATED WORK

The related efforts can be classified into four categories: fault tolerance in application layer, fault tolerance in transport layer, software maintenance, and others. We will describe these efforts in the following of this chapter.

3.1 Fault Tolerance in Application Layer

3.1.1 Recursive Restart (RR)

Recursive Restart (RR) [8][9] allows a fine-grained component-based service to restart a component (instead of the whole service) once the component fails, and thus reducing the service restart time. However, RR is not suitable for all Internet services due to the following reasons. First, the inter-component communication will degrade the system performance, which is not allowed for many Internet services. Second, RR requires an Internet service to be composed of fine-grained components, which needs redesigning the legacy Internet service programs.

3.2 Fault Tolerant in Transport Layer

3.2.1 Fine-Grained Failover System

This approach [7] aims at increasing availability of a web service which is based on a server cluster. The mechanism can be divided into two parts. First, a HTTP-aware module is inserted between the application and transport layers to log the interactions between these two layers. Moreover, this approach uses TCP migrate options to record TCP state for connection resumption. The advantage of this approach is that it requires no modification to the server

application. However, the TCP migrate options which the approach is based on requires modification to the TCP implementations of both the server and the clients, reducing the feasibility of the approach.

3.2.2 M-TCP

Migratory TCP (M-TCP) [23] allows on-line connections to be migrated from one server to another cooperative server. When a server overloads or fails, it will trigger the migration process, which makes the client to reconnect to a better performing server replica. A set of API is provided for the server application to support state transfer between server replicas. However, same as Fine-Grained Failover System, M-TCP extends both the client-side and the server-side TCP implementations to accomplish dynamic connection migration. It is difficult to deploy the TCP extension to all the clients interacting with a service.

3.2.3 TCP Splicing

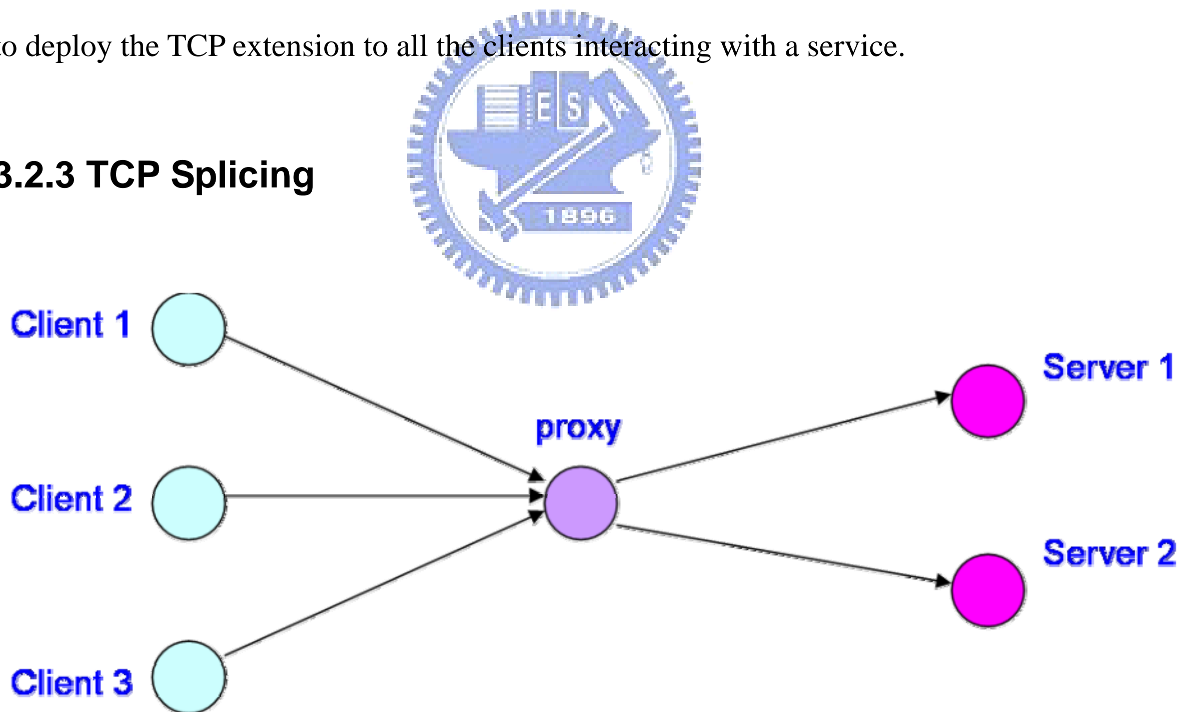


Figure 3 TCP Splicing

In this approach [17], a virtual TCP connection is composed of two physical connections, a client-proxy connection and a proxy-server connection. As shown in Figure 3, all the clients connect to a proxy, which receives and dispatches the client requests to the

back-end servers. The proxy records IP addresses, port numbers, and TCP sequence numbers of both the client-proxy and proxy-server connections. If the server crashes, the proxy can re-establish a new connection with another server and ensure that the new connection is consistent with the client's state. This approach is transparent to both the client and the server. However, it requires multiple server replicas and an extra proxy machine. In contrast, our approach does not require the proxy machine, and we address on software fault-tolerance and thus we perform recovery on a single node.

3.2.4 FT-TCP

We have described this approach in Chapter 2. FT-TCP [1] requires a primary server, a backup server, and a logger which can be co-located with the backup machine. If the primary crashes, the backup can take over the job of serving clients and replay the requests that are pending when the fault happens. As mentioned before, our approach is based on FT-TCP. However, we aim at software fault tolerance and thus perform recovery on a single node instead.

We use a virtual machine monitor, Xen [4], to consolidate the primary and the backup servers into a single physical node. However, as shown in Figure 4, simply put the servers into virtual machines is not practical since the frequent communication between two virtual machines will degrade the system performance largely. Our framework solves this problem in two ways. First, we eliminate the primary-backup communication. All the connection state logging is recorded in the memory of the VMM. As a result, the backup server VM can be suspended and the system performance can be improved. Second, we propose techniques to reduce the recovery time of FT-TCP.

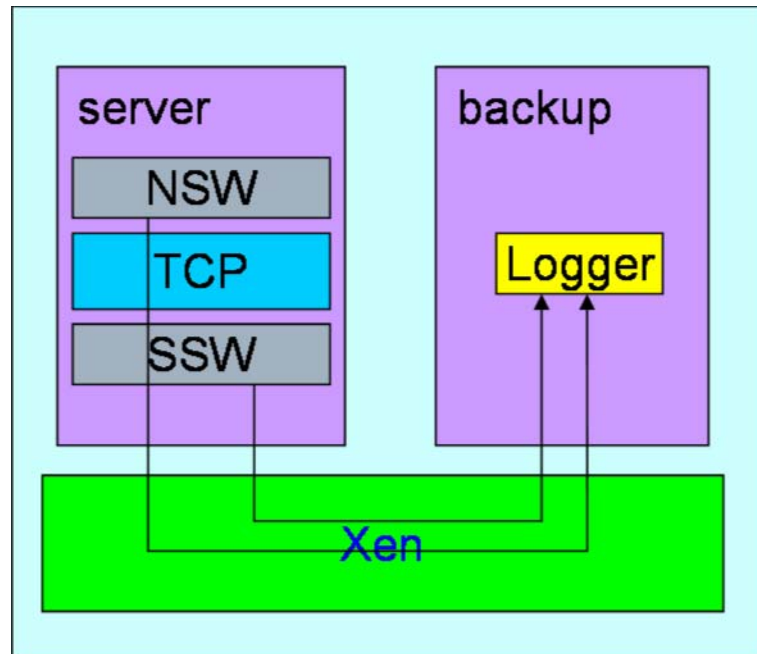


Figure 4 FT-TCP on Xen

3.3 Software Maintenance

3.3.1 Devirtualization

This approach [12] provides a strategy to boot a virtual machine monitor and a new operating system dynamically when the system needs software maintenance. Once the software maintenance operation is finished on the newly-boot operating system, the state of the old system is migrated to the new operating system. Then, the old operating system and the virtual machine monitor can be shutdown. This approach has good performance in normal operation since the VMM is only presented when maintenance is needed. However, it only focuses on decreasing the planned downtime, it can not reduce unplanned downtime.

3.4 Others

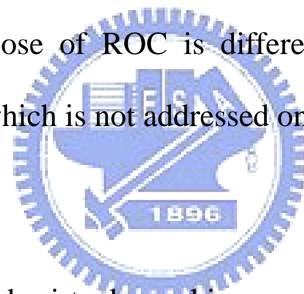
3.4.1Checkpointing

Checkpointing [16][21][22][24][25] is a common technique for system recovery. It saves the state of a running program periodically to a stable storage. When the system crashes, the

last checkpointed state can be reloaded to recover the system. This approach has two drawbacks. First, it can not solve the software aging problem since the checkpointed state is aged, instead of fresh. Even if the system can be recovered, the software may fail again immediately. Second, checkpointing usually result in large performance overhead due to the large volume of state that needs to be stored.

3.4.2 Recovery-Oriented Computing (ROC)

Recovery-Oriented Computing project [20][3] differs from the main stream of previous fault management approaches in that it concentrates on Mean Time to Repair (MTTR) rather than Mean Time to Failure (MTTF). ROC assists administrator to find out faults by its tools. After the fault occurs, it can rewind the system to a previous correct state, repair the fault, and replays the service. The purpose of ROC is different from ours. We stress on avoiding connection/service state loss, which is not addressed on ROC.



3.4.3 Xen

Xen [4] is an x86-based virtual machine monitor. It allows multiple commodity operating systems such as Linux, BSD and Windows XP to be hosted on a physical machine simultaneously. As mentioned before, we implemented our framework on Xen. The reasons are that it is open source and has little overhead.

CHAPTER 4

DESIGN AND IMPLEMENTATION

In general, functionality exported by current operating systems has the following limitations to achieve the goal of zero-loss service restart.

- Existing systems usually do not provide any mechanisms to recover the state of a service application when it crashes due to software faults. Instead, the operating system usually kills the faulty processes, which causes the internal state of the processes (including the connections being served) to be lost.
- The connection state in TCP layer will be lost when a fault crashes the service application or the operating system. For the former, the operating system will clean the connection state of the service process. For the latter, the system will be rebooted and all the system information will be lost.
- When upgrading a system, the administrator has to turn off the service, which causes the service to become unavailable for a period of time.

In this thesis, we propose a framework to overcome the above limitations.

The rest of this chapter is organized as follows. Session 4.1 explains the system components in the framework. Session 4.2 describes the proposed *fault recovery* technique. We will explain how to recover a service when a fault occurs. Session 4.2 describes the proposed *online maintenance* technique. We will explain how to avoid the downtime caused by system maintenance.

4.1 System Components

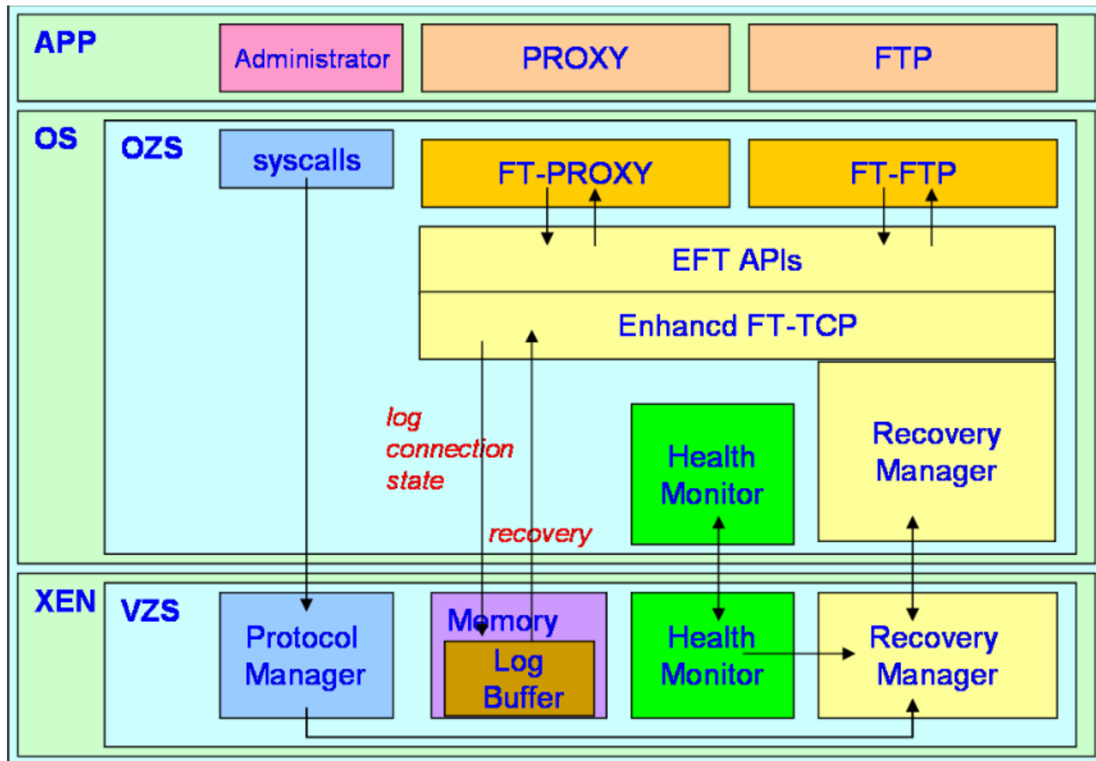


Figure 5 Our System Components

As shown in Figure 5, in order to achieve the goals mentioned above, we implement our framework in both the operating system kernel (i.e., Linux) and the virtual machine monitor (i.e., Xen). The former part is called *OS layer Zero-loss Subsystem (OZS)* while the latter part is called *VMM layer Zero-loss Subsystem (VZS)*.

The major components of the framework are: protocol manager, health monitor and recovery manager. In addition to the components, we also enhance FT-TCP to reduce the recovery time and provide an API for the service designers to develop their fault tolerant service. Moreover, the framework provides system calls for the administrators to control the backup server and the service migration.

4.2 Fault Recovery

Briefly speacking, we use four techniques to achieve the goal of fault recovery. First, we develop a protocol to create/suspend/resume the backup server. During normal operation, the backup server is suspended so that it does not contend CPU resources with the primary server. Once the primary fails, the backup server is resumed to take over the job of the primary. Second, we provide a log buffer in VMM which allows us to store the connection state without communicating with the backup server. Third, we provide a fault detection mechanism, which can detect application or operating system faults and then trigger the recovery job. Finally, we provide a recovery mechanism to recover the service state.

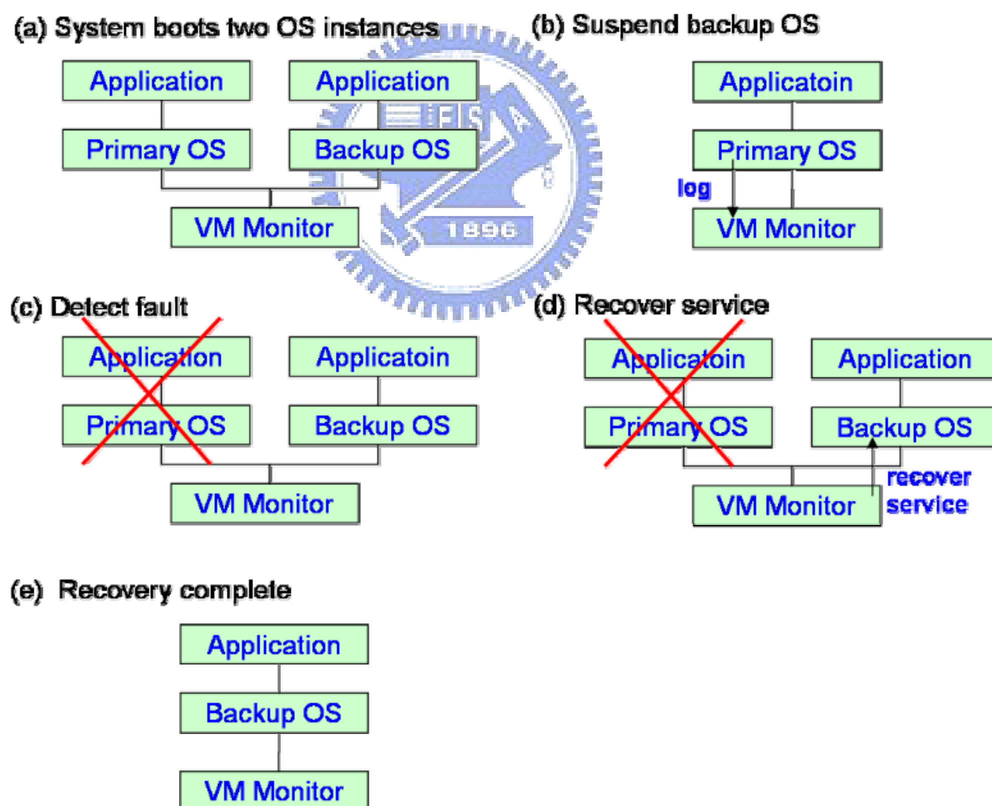


Figure 6. An Overview of Fault Recovery

We give a brief overview of the recovery flow first, which is shown in Figure 6, before the detailed description of our fault recovery techniques. Before starting an Internet service, the administrator starts a backup server, including the backup OS and service application.

Then, in order to supply the primary server with the whole system resources, the backup server releases the resources such as CPU time it holds. The primary server then does the normal operations and logs the connection information in the Virtual Machine Monitor (VMM). When a fault is detected, VMM wakes up the backup server and recovers the service state so that the system can provide the service continually.

In the following, we will describe the details of the techniques. Section 4.2.1 describes the way to boot a second OS instance and release the system resource used by the second OS instance. The flow of logging connection state is presented in Section 4.2.2. Section 4.2.3 describes the fault detection mechanism, and the recovery flow is presented in Section 4.2.4.

4.2.1 Backup Server Boot-up

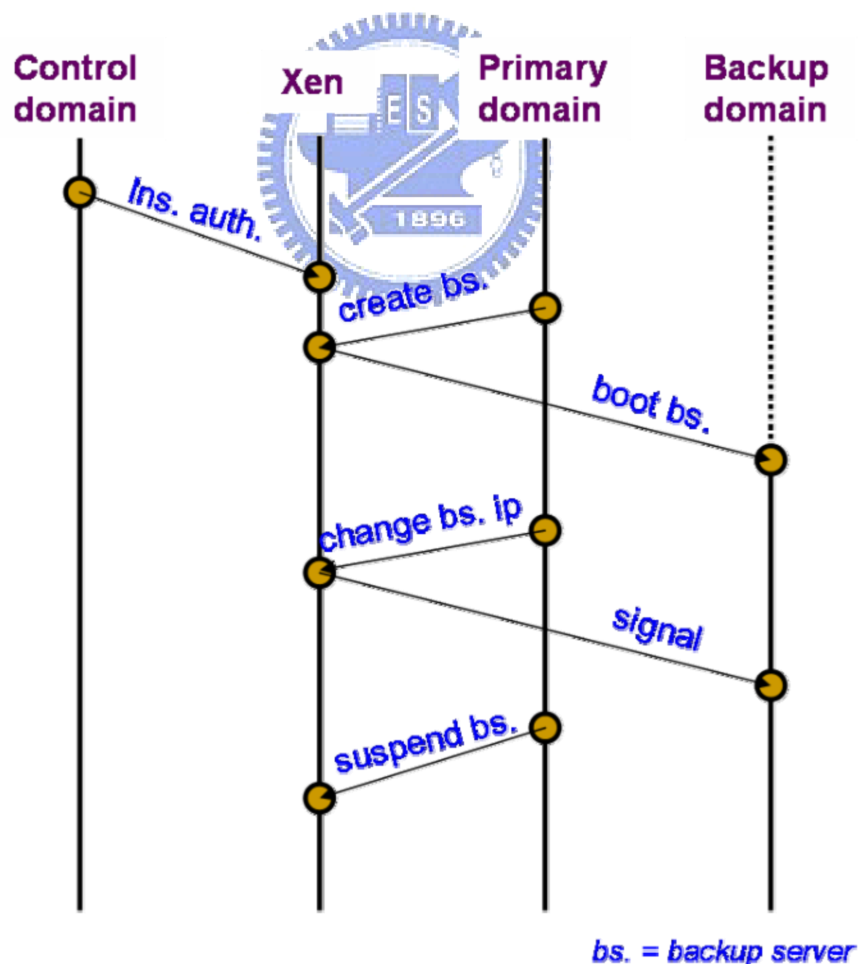


Figure 7 The Flow of Booting a Backup Server

We define a protocol to manage the boot up of the backup server. The protocol involves the VMM and three domains: control, primary and backup, which implement the protocol based on the API, as shown by Table 1, provided by the framework.

Table 1 System Calls Provided by OZS

syscall table		
Function name	Source/Dest.	Description
sys_ins_auth()	con/vmm	Install authentication info. to vmm
sys_boot_backup_server()	prim/vmm	Boot backup server
sys_change_backup_ip()	prim/backup	Change backup ip address
sys_suspend_backup_server()	prim/vmm	Suspend backup server
sys_wakeup_backup_server()	prim/vmm	Wake up backup server
sys_migrate_service()	prim/vmm	Notify recovery manager to migrate services

1896

Figure 7 shows the flow of booting up a backup server. Originally, Xen only allows the control domain to boot up other domains. In order to enable an authorized primary server to boot up its backup, we allow the administrator to register the primary servers that has the right to boot up their backups. Specifically, the administrator can register an entry for each primary server that has that right in the *backup-grant* table in advance. The table is stored in VMM and managed by the protocol manager, and the registration is done by calling the *sys_ins_auth()* system call in the control domain. When a primary server boots a backup server, the protocol manager will check if the primary server has the grant.

The primary server calls the *sys_boot_backup_server*() system call to ask Xen to create the backup. As mentioned above, the protocol manager checks to see if the primary server is granted to boot its backup. If it is, the protocol manager asks Xen to create the backup domain .

Originally, Xen gives an unique IP address to each guest OS so that each domain can communicate with external machines. This results in a longer recovery time since the backup

server has to take over the IP address of the primary server when the latter crashes. Thus, we provide a *sys_change_backup_ip()* system call to allow the primary and backup servers to share a single IP address. When the system call is invoked by the primary server, a signal will be sent to the backup server through the VZS, and the backup server will get the primary IP address from the VZS and change its IP address accordingly. The IP address changing is done by a user-level task which invokes a shell command - *ipconfig*.

After the IP address is changed, the backup server should release its CPU time so that it will not affect the performance of the primary server. This is done by calling the *sys_suspend_backup()* system call by the primary server. When the system call is invoked, Xen will remove the backup server task from the run queue of Xen.

From the above description we can see that, although the system calls are implemented in the OZS, most of them require cooperation from the VZS. The communication between OZS and VZS is through hypercalls and events.

4.2.2 Connection State Logging

FT-TCP provides a log buffer to record the connection state of the primary server. When the primary server crashes, the backup server will use the data in the log buffer to recover the system. In our design, we also provide a log buffer which does not lose data even when the primary server crashes. We use a memory area of the primary server as the logger buffer. During the recovery period, backup server will remap the log buffer into its virtual address space and recover the service state accordingly. In the following, we describe how to implement the log buffer in our framework.

In order to let guest operating systems manage memory conveniently, Xen provides an illusional memory area, a continuous range of physical addresses, for each guest OS. However, physical address is not real machine address. Therefore, there are two problems deserving to be mentioned. First, as mentioned above, the backup server has to map the log buffer into its

virtual address space. This mapping requires the starting machine address of the log buffer. However, a guest OS does not manage machine addresses directly. Thus, we lookup the page table of the guest OS, which is updated by Xen, to get the machine address of the log buffer. Once the address is obtained, the OZS issues a hypercall to Xen in order to register the address. As a result, the backup server can get the machine address of the log buffer during the recovery period.

Second, if a primary server crashes, its memory area (including the log buffer) will be released by Xen. To avoid releasing the memory before recovering the service, we increase the reference count that corresponds to the primary server by 1 after booting the primary. After the service recovery, the reference count is decreased by 1 and the resources held by the primary server can be released.

4.2.3 Fault Detection

Software faults, which cause the system become unavailable, can happen on service applications and the operating system. In the following, we describe how to detect the faults.

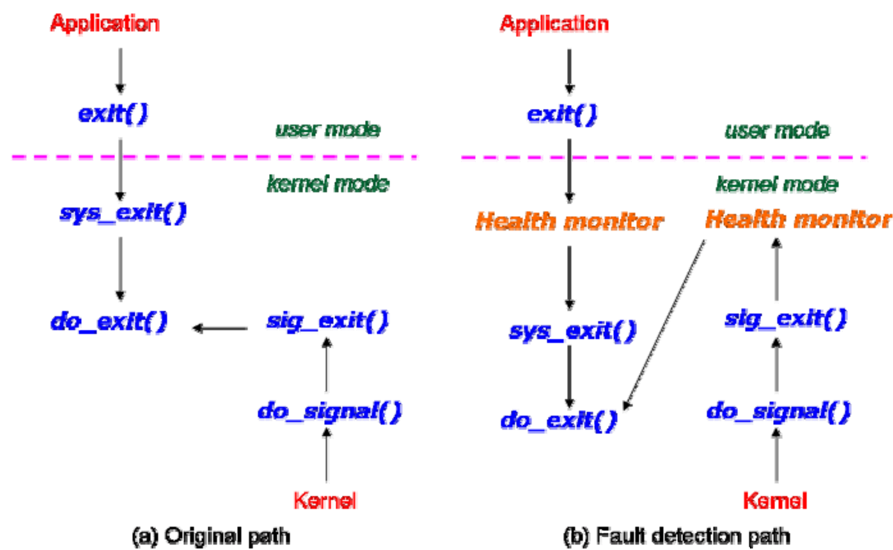


Figure 8 Detecting Application Faults

When a fault occurs on an application, the kernel usually invokes the *do_exit()* function

to kill the application process. As shown in Figure 8(a), two paths lead to the invocation of *do_exit()*. One is that application detects the fault itself and calls the *sys_exit()* system call, which in turn calls *do_exit()*. The other is that kernel detects the application fault and sends a signal to kill the application process. In this case, kernel calls *do_exit()* through *sig_exit()*. Originally, we can intercept *do_exit()*, by kernel binary instrumentation, to detect the faults. However, such callee-based instrumentation requires more efforts. Therefore, we use the caller-based instrumentation approach instead. As shown in Figure 8(b), the *health monitor* intercepts the *exit()* system call and the *sig_exit()* function, which only requires modifying the destination addresses of two jump instructions.

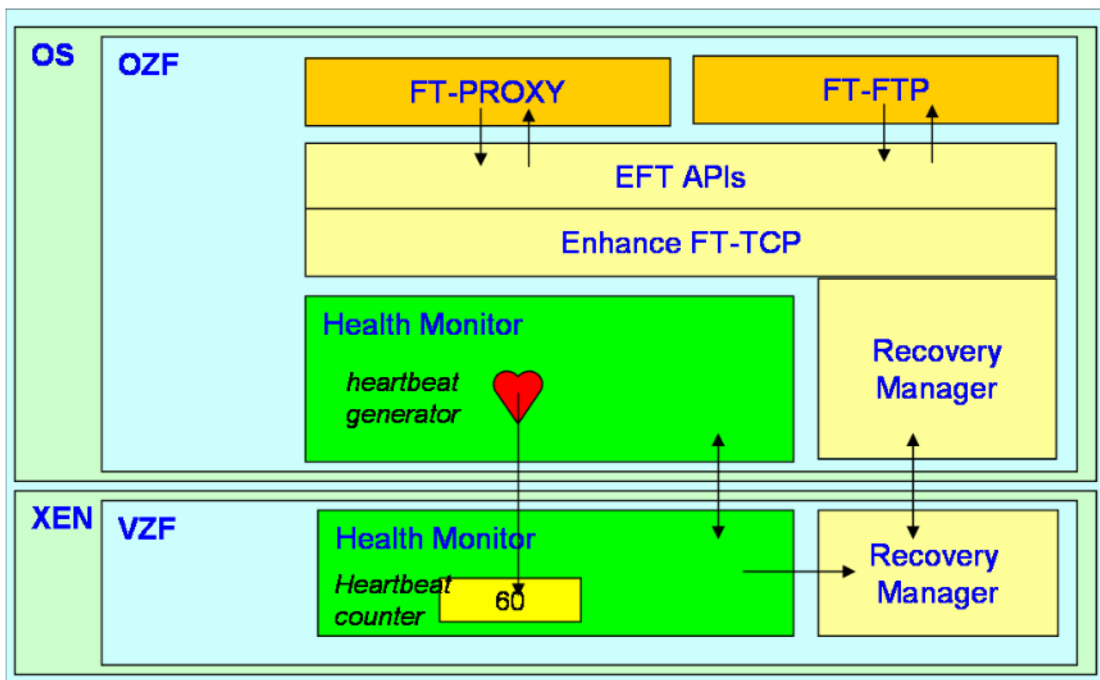


Figure 9 Detect Kernel Fault

In addition to application faults, operating system faults may also occur. To detect such faults, we inserted a *heartbeat generator* in the primary server domain and a heartbeat checker in Xen. At each timer interrupt, the former sends a heartbeat to Xen by increasing the value of the *heartbeat counter* variable by one, which is shared by the primary server domain and Xen.

The latter checks the variable at each timer interrupt to detect operating system faults. If the value remains the same during two timer interrupt periods, the operating system is regarded as failure, and the checker notifies the *recovery manager* to recover the system. It is worth noting that the heartbeat mechanism is implemented based on shared memory instead of hypercall, and thus it eliminates the overhead of frequent privilege mode crossings.

4.2.4 Recovery Flow

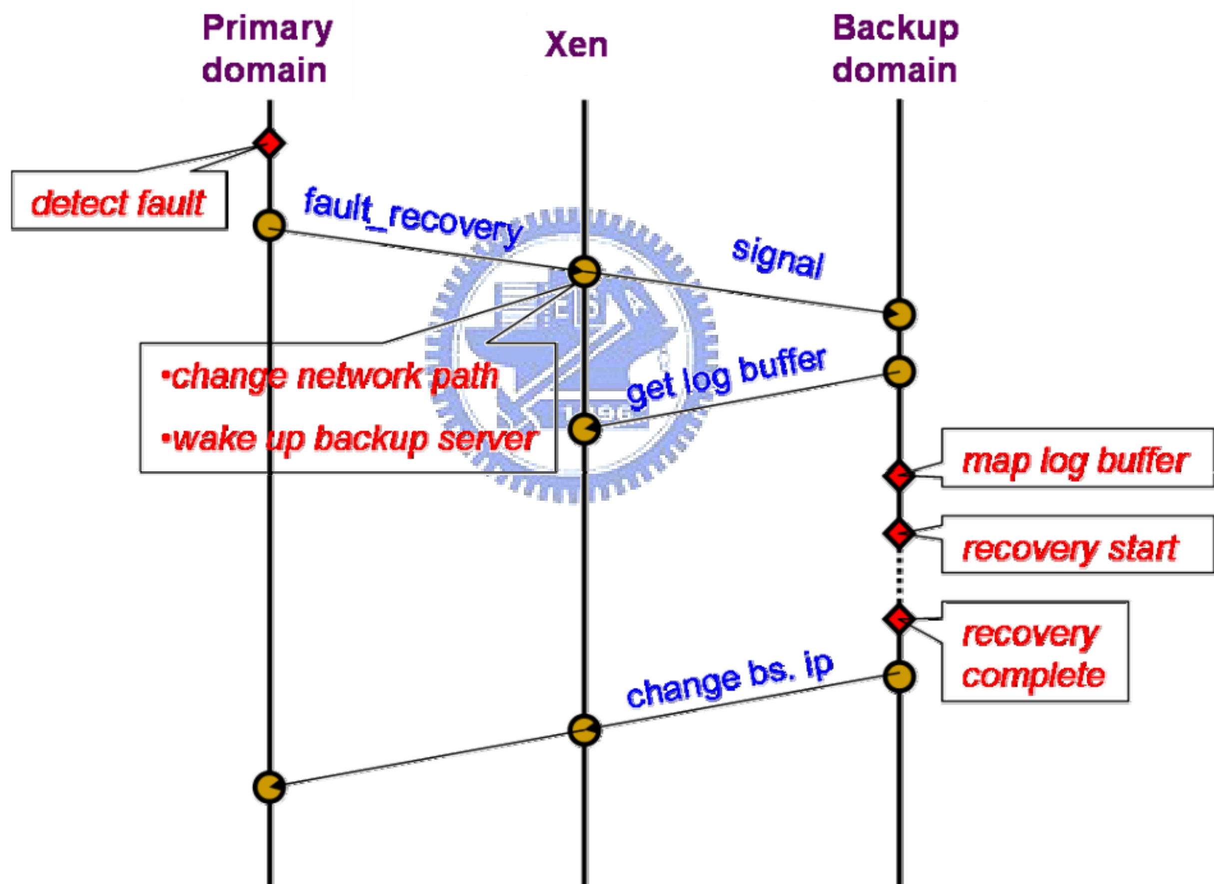


Figure 10 Recovery Protocol

When a fault is detected, the recovery manager will follow the recovery protocol to recover the system. Figure 10 illustrates the recovery protocol, which is divided into three steps. First, the recovery manager must change the network path so that incoming packets

which are originally delivered to the primary server will now be delivered to the backup server. Xen stores IP-to-domain mappings for each domain (i.e., in the *net_schedule_list* list) in order to perform packet delivery, and thus the network path changing can simply be done by updating the mapping that corresponds to the IP address of the backup server. Second, the recovery manager must wake up the backup server so that the backup server can take over the job of the primary server. Third, the recovery manager must send a signal to notify the backup server to recover the system. When receiving the signal, the kernel subsystem in the backup server will obtain the machine address of the log buffer through a hypercall, remap the log buffer, and then execute the FT-TCP recovery flow.

It is worth mentioning that, if the fault does not crash the kernel of the primary domain, we can change the IP address and the packet delivery path (in Xen) so that a system administrator can connect to the faulty server to diagnosis the reason of the fault.

4.3 Online Maintenance

To allow online maintenance, we use some mechanisms that are the same as those we use for fault recovery. For example, we also provide a backup server and a log buffer. However, we add a functionality to allow online maintenance. Specifically, we provide a *sys_migrate_service()* system call(as shown in Table 1), which is used to migrate an Internet service when the administrator completes system maintenance. We will describe it in the next section.

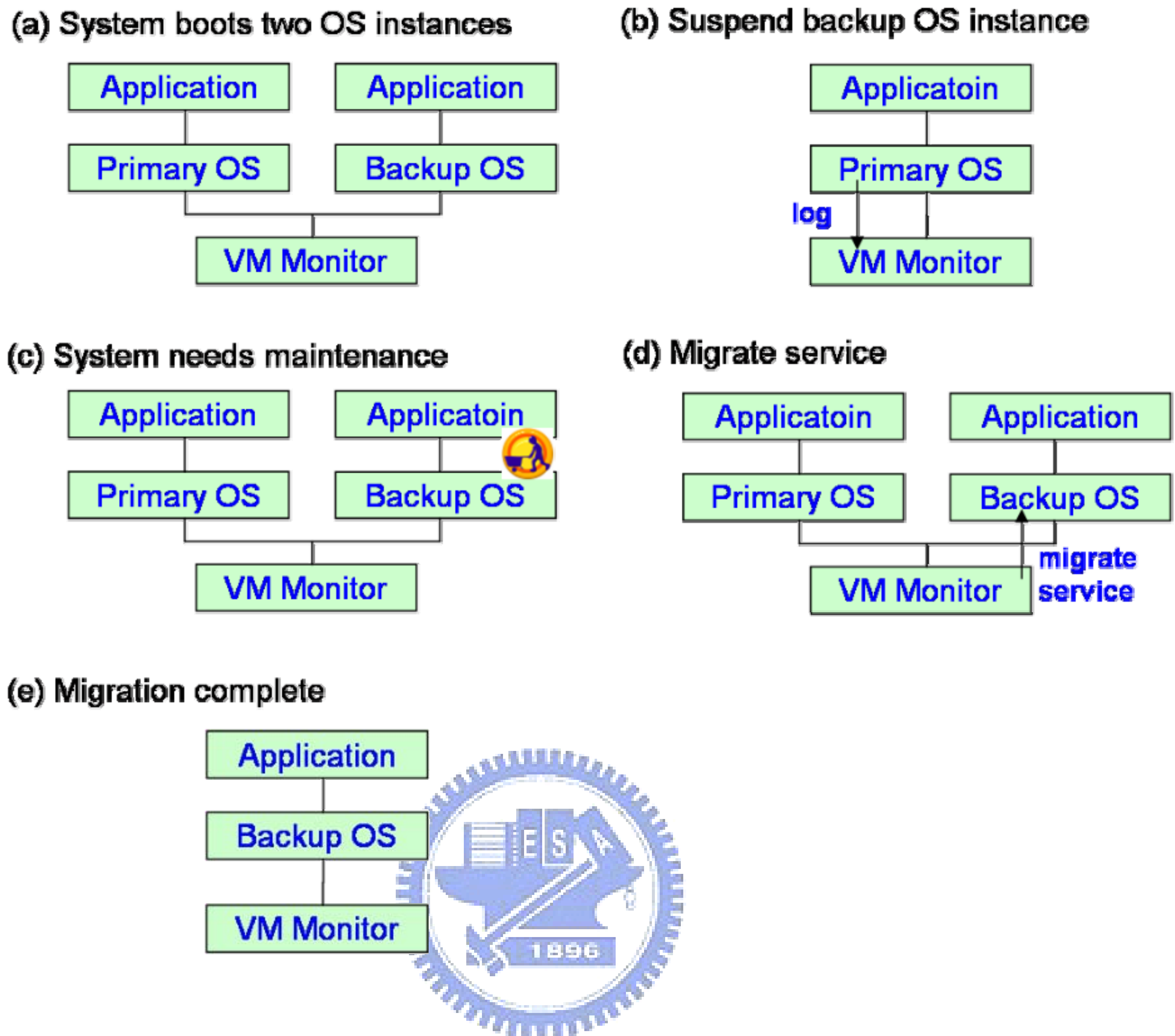


Figure 11 An Overview of the Online Maintenance Flow

Figure 11 shows a brief overview of the online maintenance flow. Firstly, as shown in Figure 11(a) and (b), the OZS boots a backup server and then suspend it. When system needs maintenance, the kernel subsystem wakes up the backup server, and the administrator can upgrade the system on the backup server, as shown in Figure 11(c). When the system maintenance is completed, the administrator can use the *sys_migrate_service()* system call to migrate the service state from the primary server to the backup server, as shown in Figure 11(d). Finally, as shown in Figure 11(e), the backup server takes over the job and the clients can continuously be served without interruption caused by system maintenance.

4.3.1 Maintenance Flow

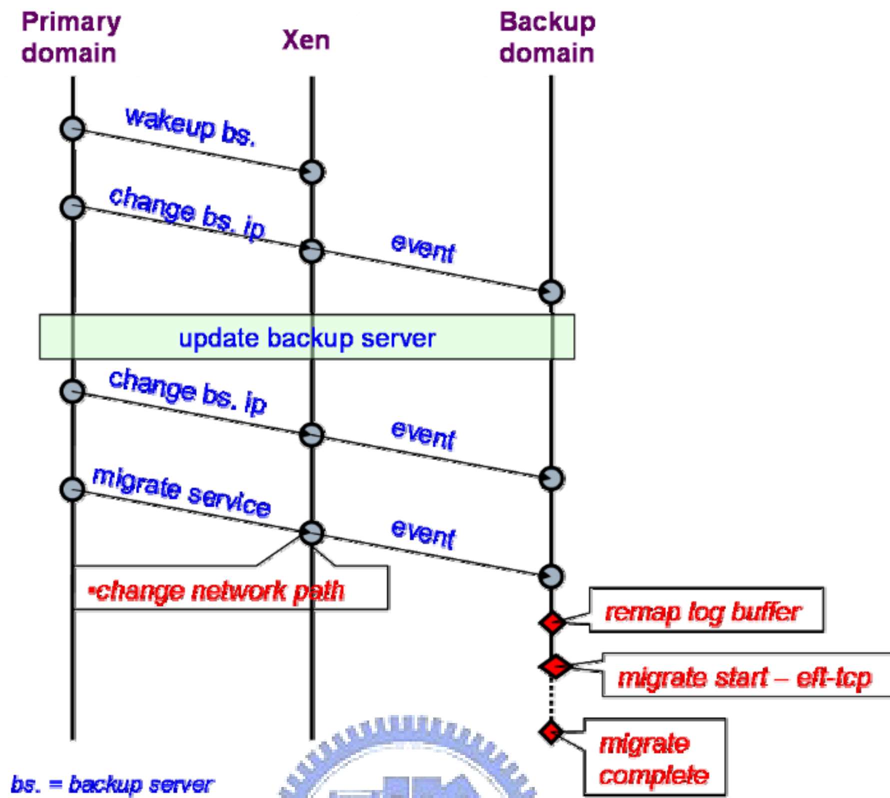


Figure 12 The Flow of Service Migration

1896

Figure 12 shows the flow for achieving online maintenance. As mentioned above, the backup server suspends itself after it has initialized, and the IP address of the backup server has been changed to the IP address of the primary server in order to allow fast fault recovery. Therefore, to allow online maintenance, the administrator first wakes up the backup server and restore its IP address by calling the sys_wakup_backup_server() and sys_change_backup_ip() system calls, respectively. When the system maintenance on the backup server is finished, the administrator changes the IP address of the backup server to that of the primary server and calls the *sys_migrate_service()* system call(as shown in Table 1) to migrate the service. This system call notifies the recovery manager to migrate the service through the protocol manager. The recovery manager will use the strategy mentioned in section 4.2.4 to migrate the service.

CHAPTER 5

RECOVERY TIME REDUCTION

For some Internet services, FT-TCP may take a long time to recover them. We will describe these conditions in the following section. Therefore, we provide an API for Internet services to reduce the recovery time. In the rest of this section, we take two Internet services (FTP and HTTP proxy) as examples to demonstrate how to reduce the recovery time by using the API.

5.1 FTP

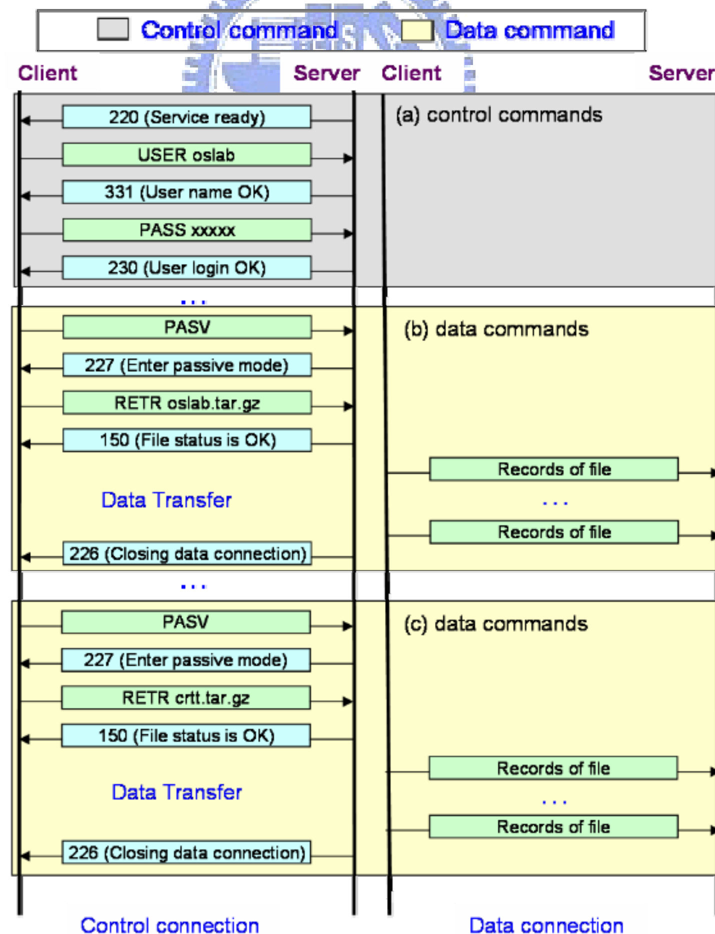


Figure 13 A Example of Using FTP for Sending Files

Many Internet service protocols, such as FTP, SAMBA, and HTTP 1.1, uses long-lived connections/sessions for object transmission. FT-TCP may take a long time when recovering such services since it replays the connection reestablishment and object transmission from the beginning. We illustrate such condition with an FTP example. As shown in Figure 13, the FTP session contains control commands and a sequence of data commands. If a fault happens during the transmission of the crtt.tar.gz file (i.e., the second data connection), FT-TCP will re-establish all the control and data commands. However, as shown in Figure 13, the firs data commands is not needed to be recovered since it has completed successfully before the fault happens. Removing such data commands from the recovery job could improve the service recovery time.

Table 2 Wrapper Registration APIs

Wrapper Registration APIs	
Function name	Usage
wr_create()	Create wrapper structure. We also use this api to install application's port.
wr_ins_nsw()	Install north-side wrapper
wr_ins_ssw()	Install south-side wrapper
wr_unins_nsw()	Uninstall north-side wrapper
wr_unins_ssw()	Uninstall south-side wrapper

Table 3 Application-Specific Hook APIs

Application-Specific Hook APIs	
Function name	Usage
as_remove_request()	Remove some data commands that are completed successfully.
as_reomove_response()	Remove some response length of successful completed data commands.

Table 4 Client-Side APIs

Client-Side South Side Wrapper APIs	
Function name	**Usage**
cs_rec_syn()	When a server receives syn. packet, SSW will initial some values of log buffer through this function.
cs_rec_ack()	When a server receives ack. packet, SSW will record ack seq. and packet payload during normal operation. Moreover, SSW will modify packet during recovery.
cs_rec_fin()	When a server receives fin packet, SSW will prepare to close a connection through this function.
cs_rec_rst()	When a server receives rst packet, SSW will close this connection through this function.
cs_snd_ack()	When a server sends ack packet, SSW will record ack seq during normal operation. And, SSW will modify packet during recovery.
cs_snd_saa()	When a server receives syn and ack packet in recovery state, SSW will calculate delta number during recovery.
cs_snd_fin()	When a server sends fin packet, SSW will prepare to close a connection through this function.
cs_snd_rst()	When a server receives rst packet, SSW will close this connection through this function.
Client-Side North Side Wrapper APIs	
Function name	**Usage**
cn_read_request()	When a service reads the request, NSW will record the returned value during normal operation. Moreover, NSW will provide fake request during recovery.
cn_write_response()	When a service writes the request, NSW will bypass during normal operation. And, NSW will intercept response that had sent during recovery.

Applying such optimization on FT-TCP requires digging into the FT-TCP code and modifying it. Moreover, such optimization may only be suitable for a special class of services, such as FTP and HTTP1.1. Therefore, in order to reduce the effort of the developers, we decompose FT-TCP into several basic operations and export a function for each operation. Thus, when implementing a fault tolerant service, the developers can invoke the operations provided by FT-TCP according to their needs. Table 2, Table 3, and Table 4 show the exported functions. The functions that correspond to the original FT-TCP implementation can be divided into two categories. The first category is *Wrapper Registration* APIs, by which designers can install wrapper function. The second category is *Client-side South-side Wrapper* (CSW) API and *Client-side North-side Wrapper* (CNW) API, by which designers can log and recover TCP state, and record interactions between TCP and service application so as to recover the service state respectively. We added the third category (i.e., *Application-Specific*

API), which is add-on functionality for some Internet services. For example, we implemented a function *as_remove_request()* for fault tolerant FTP developers to remove requests that are not needed to be replayed during the fault recovery period.

At the last of this section, we describe how to use the API to implement a fault tolerant FTP system. First, the developers should register handlers that will be invoked when a packet reaches the NSW or the SSW (i.e., wr_ins_nsw() and wr_ins_ssw()). Then, they should invoke the API shown in Table2. For example, if a SYN packet is received, they can invoke the cs_rec_syn() to log the packet. Finally, the FTP developers should record how many data connection commands the server has received such as PASV, LIST, RETR, and remove the commands that correspond to a transmission-completed file. After transmitting a file, the FTP server will send a 226 command to the client, and hence the FTP developers can call *as_remove_request()* to remove the data connection commands accordingly.

5.2 HTTP Proxy

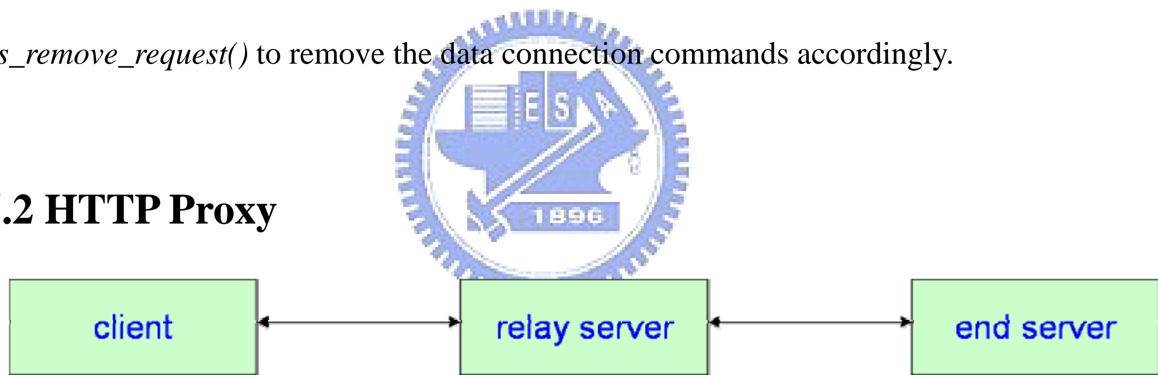


Figure 14 Relay Server

As shown in Figure 14, some Internet servers such as proxies and email servers act as both clients and servers, which are called *relay servers*. The others servers only play the server role, which are called *end servers*.

FT-TCP can recover a system in a client transparent way. However, the recovery can not be done in a server transparent way. Therefore, it is not suitable for relay servers. Specifically, applying FT-TCP on relay servers would cause the server to establish a number of new connections with the end servers once the service on the relay server crashes. This results in

long recovery time and may cause data inconsistency for dynamic-object requests or transaction based connections. Therefore, we extend the FT-TCP implementation to achieve server transparency and avoid such connection establishment. A fault tolerant relay service can use the API shown in Table 3 to achieve sever-side transparency.

Table 5 Server Side APIs

Server-Side South Side Wrapper APIs	
Function name	**Usage**
ss_snd_syn()	When a server sends syn. packet, SSW will initial some values of log buffer through this function.
ss_snd_ack()	When a server sends ack. packet, SSW will record ack seq during normal operation. And, SSW will modify packet during recovery.
ss_snd_fin()	When a server sends fin packet, SSW will prepare to close a connection through this function.
ss_snd_rst()	When a server sends rst packet, SSW will close a connection through this function.
ss_rec_ack()	When a server receives ack packet, SSW will will record ack seq. and packet payload during normal operation. Moreover, SSW will modify packet during recovery.
ss_rec_saa()	When a server receives syn and ack packet in recovery state during recovery, SSW will calculate delta number.
ss_rec_fin()	When a server receives fin packet, SSW will prepare to close a connection through this function.
ss_rec_rst()	When a server receives fin packet, SSW will close a connection through this function.
Server-Side North Side Wrapper APIs	
Function name	**Usage**
sn_read_response()	When a service reads the response, NSW will record the returned value during normal operation. Moreover, NSW will provide fake data during recovery.
sn_write_request()	When a service writes the request, NSW will bypass during normal operation. And, NSW will intercept request that had sent during recovery.

This API is a counterpart to the client side API As shown in the Table 5, the API can be divided into two categories. The first is *Server-side South-side Wrapper* (SSW) API, by which developers can log and recover TCP state. The second is *Server-Side North-side Wrapper* (SNW) API, by which developers can record interactions between TCP and the service application so as to recover the service state. They are similar to functionality of FT-TCP.

CHAPTER 6

PERFORMANCE EVALUATION

In this section, we evaluate the effectiveness and efficiency of our framework. We implemented a fault tolerant proxy (i.e., ft_proxy) and a fault tolerant FTP server (i.e., ft_ftp) based on our framework, and we measure their performance with or without the presence of faults.

6.1 Experimental Environment

Table 6 Experimental Environment

	client	FT machine	end server
Hardware	P4 1.6 GHz, 256MB memory	P4 2.0 GHz, 1GB memory	P4 2.0 GHz, 256MB memory
VMM	N/A	Xen 1.2	N/A
OS	Linux 2.4.18	xenolinux 2.4.26	Linux 2.4.18
Guest os	N/A	control domain-64MB Primary domain- 256MB Backup domain-256MB	N/A
software	WebStone 2.5, dkftpbench 0.45 wget	Squid-2.5.STABLE4, Proftpd-1.2.8	Apache 2.0.40

As shown in Table 6, we run the experiments by using three machines, one for clients, one for the fault tolerant system (i.e., the FT machine) and the other for the end server, which are connected via 100MbpsFast Ethernet links. The client machine runs Linux kernel 2.4.18 and benchmark applications such as Webstone version 2.5, dkftpbench 0.45 and wget. The end server machine runs Linux kernel 2.4.18 and an Apache server, version 2.0.40. The FT machine runs Xen 1.2 as the virtual machine monitor, xenolinux 2.4.26 as the guest operating

system, and Squid-2.5.STABLE4 and a Proftpd-1.2.8 as the applications. Our proxy server is an Intel Pentium 4 2.0 GHz PC with 1GB of memory, while both the web server and the client run on Pentium 4 2.0 GHz PC with 256MB of memory and Pentium 4 1.6 GHz PC with 256MB of memory respectively. We give 64MB, 256MB and 256MB of memory for control domain, primary domain and backup domain respectively.

6.2 Overhead

6.2.1 FT-TCP Overhead

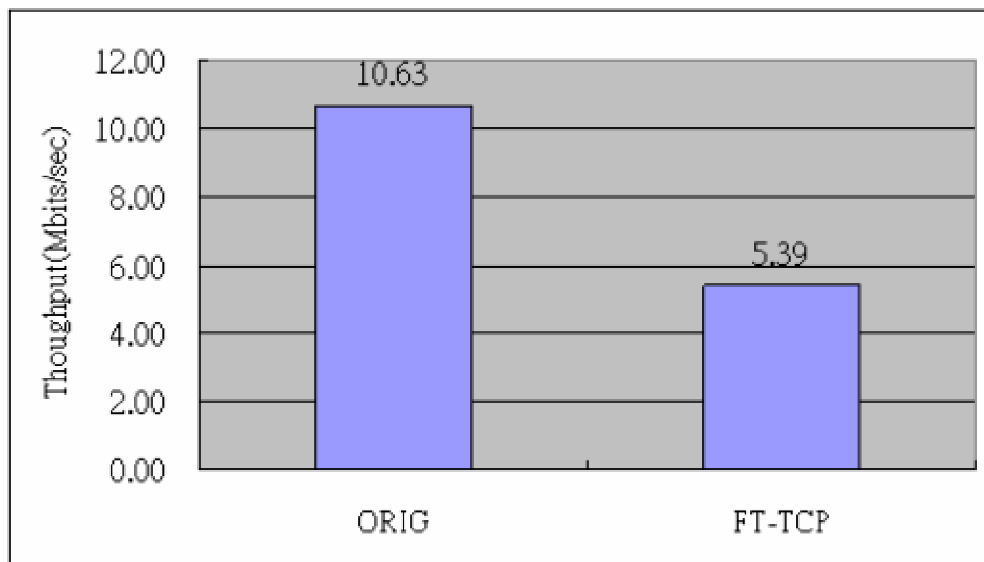


Figure 15 Performance of FT-TCP

In order to prove FT-TCP architecture is not suitable in VMM, we implement a FT-TCP architecture in Squid. We use two domains in Xen, one runs as primary server and another rus as backup server. The primary will send log data to backup server. We download a 5MB file twenty times by wget in original squid and squid which used FT-TCP architecture. As shown in Figure 15, the y-axis represents throughput, which is the mean of tests of twenty times. We can find that squid which used FT-TCP architecture results in 49% overhead degradation.

6.2.2 Squid Performance Overhead

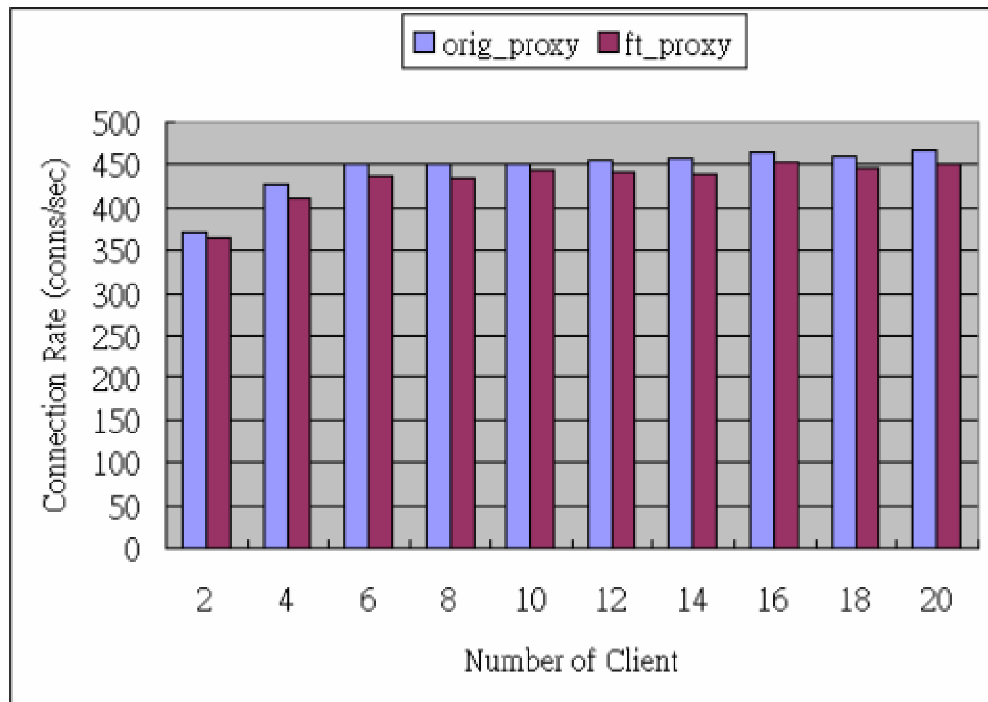


Figure 16 Performance of Squid (Connection Rate)

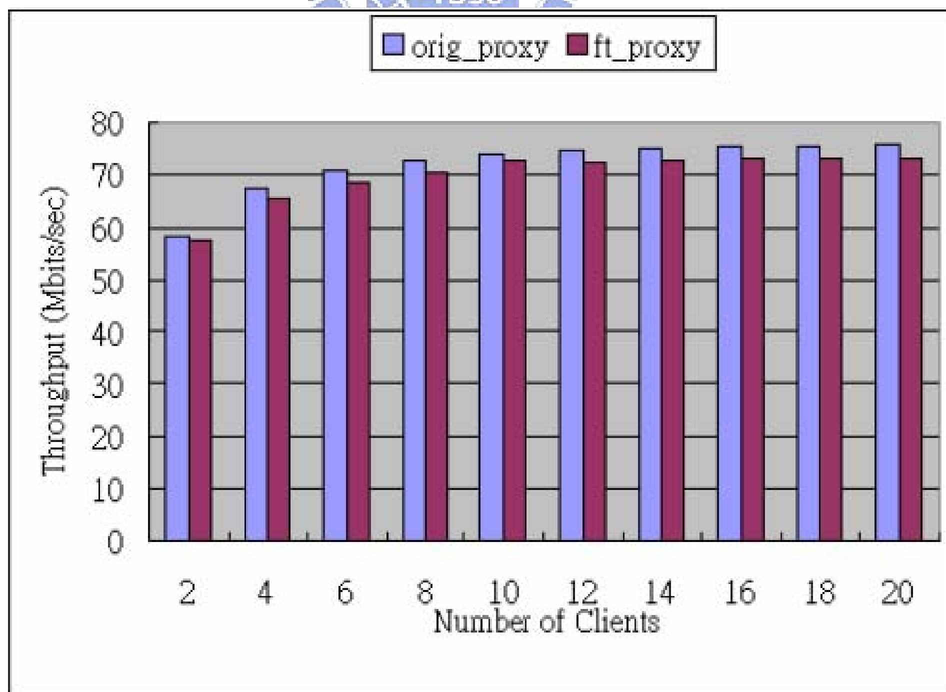


Figure 17 Performance of Squid (Connection Throughput)

Figure 18 Performance of Squid (Response Time)

We use the standard workload of WebStone benchmark version 2.5 to measure the impact of states logging on Squid performance. The benchmark simulates that many clients connect simultaneously and make requests to a web server within a defined time. Webstone could analyze the average connection rate, average connection throughput and average response time from the simulated clients. We set the increased client numbers at the WebStone. Each client number runs five minutes to test overhead of our framework.

Figure 16, Figure 17 and Figure 18 show the performance comparison between the original squid and squid used our framework. The x-axis represents the number of clients simulated by WebStone. Each client establishes a large number of connections with the server during the experiment. The y-axis indicates the server connection rate, connection throughput and response time respectively. The dark bars present the performance of Squid running on the normal operating system and the light bars present the performance of Squid running on our framework. From the figure we can see that, using our framework results in little throughput degradation that ranges 1% to 4%. This shows that our framework is quite

efficient.

6.2.2 Proftpd Performance Overhead

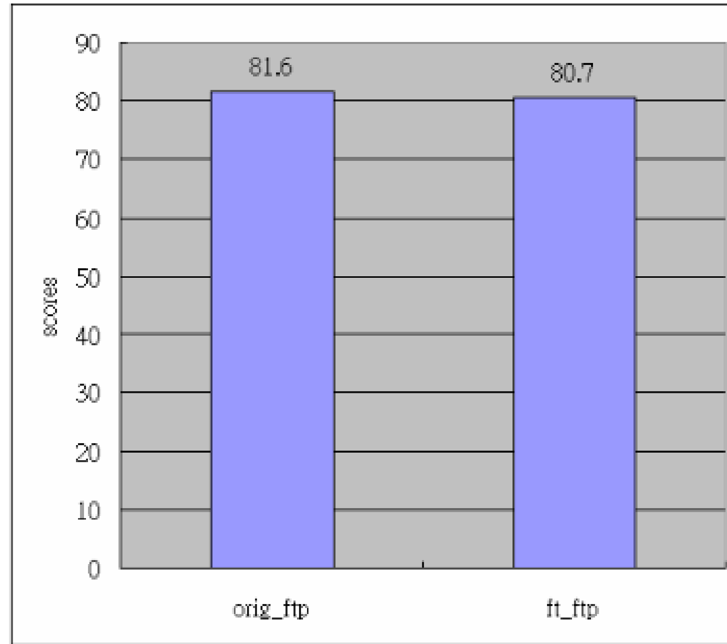


Figure 19 Performance of Proftpd (Connection Throughput)

In addition to the performance overhead of Squid, we also measure performance overhead of Proftpd. We use dkftpbench 0.45 to measure the impact of Proftpd running on our framework. This benchmark is run repeatedly with different numbers of simulated users to determine the maximum number that can be supported. We define a minimum quality of service that simulated users must receive 100KB/s. If the simulated users reach the throughput, ftp server get one score. The dkftpbench would show total score by aggregating clients whose bandwidth excess 100KB/s. We compute an average total score through running 10 times. Figure 19 show the score comparison between the original proftpd and proftpd used our framework. The y-axis indicates the scores reported by dkftpbench. From the figure we can see that using our framework results in about 1.12% throughput degradation.

6.3 Recovery Time

6.3.1 Squid Recovery Time

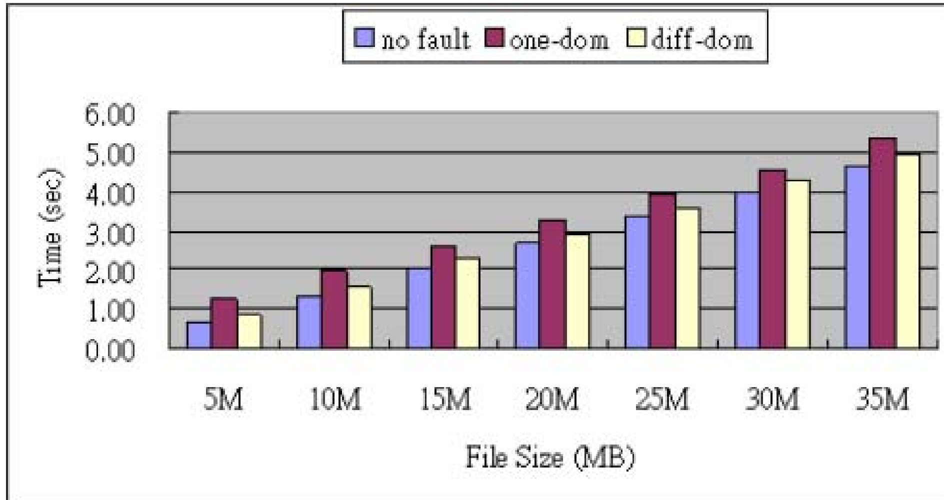


Figure 20 Fault Occurs When the First 10KB of Data is Sent.

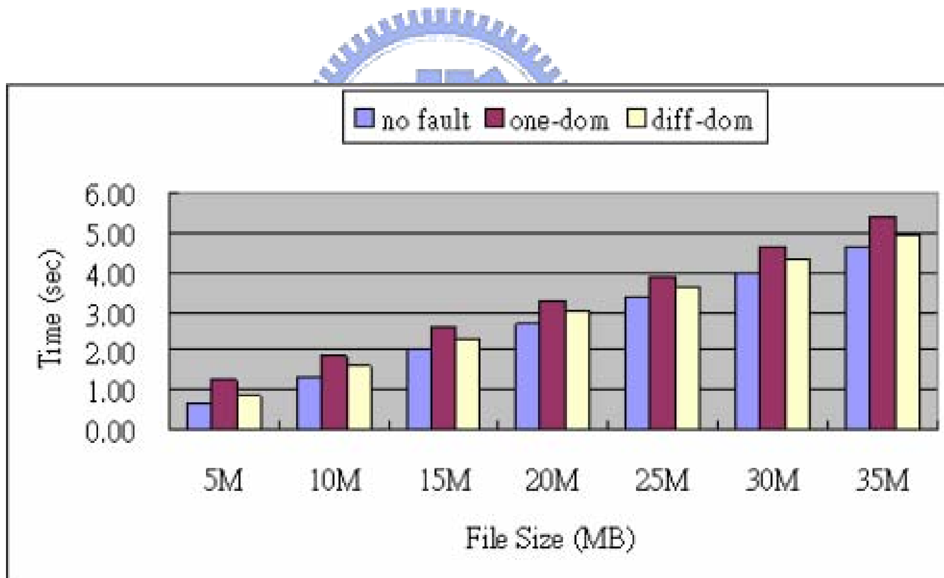
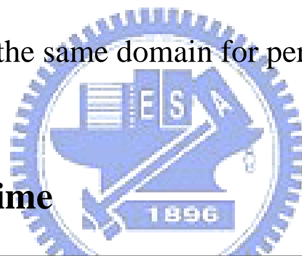


Figure 21 Fault Occurs When the First Half of Data is Sent

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. Squid process is being terminated intentionally when the client receives the first 10KB of data and the first halt of data in each run. The transmission time is measured in Fast Ethernet

environment. Figure 20 and Figure 21 show transmission time which the fault occurs when the first 10KB of data and the first halt of data is sent respectively. The x-axis stands for file size which is sent to the client from the server and the y-axis indicates the transmission time. These three lines mean the transmission time sending the different file size in the different conditions. The blue line represents the no fault condition. The red line represents that a fault occurs when sending a file and recover the service in the different domain. The green line represents that a fault occurs and recover the service in the same domain. According to the result, the recovery latency is about 250ms and 600 ms in the green line and the red line respective. We can know that recovery in the different domain is more efficient. The reason is that recovery in the same domain must wait squid to restart. It takes about 300ms. Therefore, when transient fault or software aging problem occurs, we should recover service in the different domain rather than in the same domain for performance consideration.


E S A
1896

6.3.2 Proftpd Recovery Time

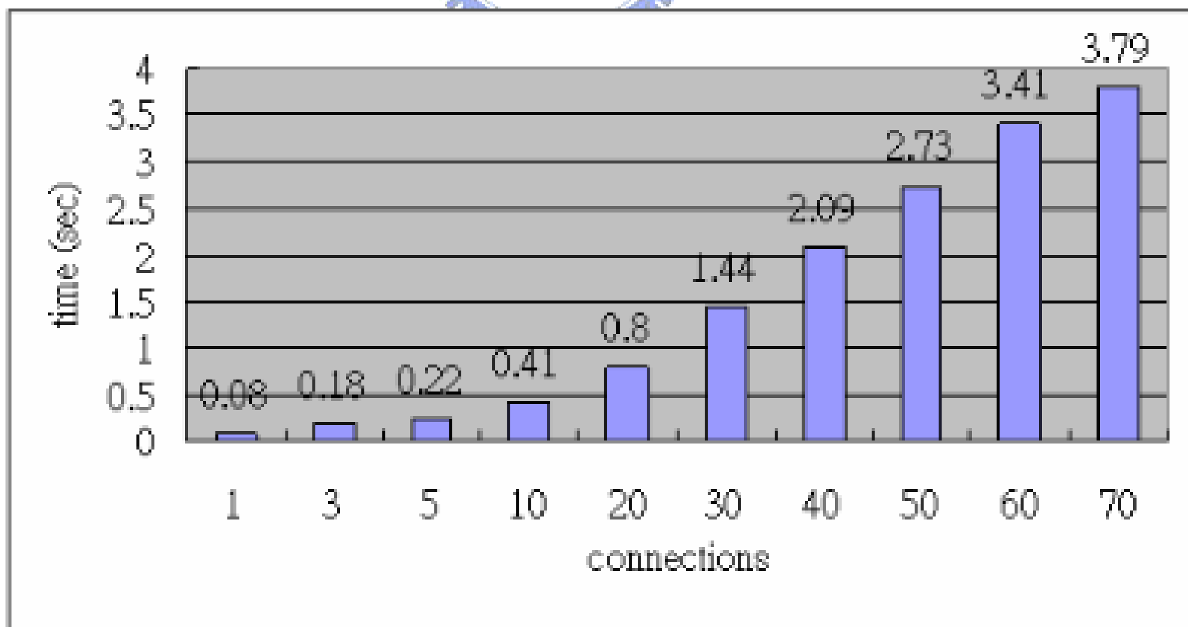


Figure 22 Relation between Number of Connections and Recovery Time

Figure 22 shows the relation between number of connections need to recover and the total recovery time. Each client connection send six control command requests (include USER, PASS, SYST, PWD, TYPE I, and CWD) and two data command requests (include PASV and RETR) to request a 20MB file. We inject a fault when the last data connection sent the first 1MB file. Obviously, recovering 70 connections only takes 3.79 sec. The recovery latency is acceptable.

CHAPTER 7

CONCLUSION

In this thesis, we propose a framework that achieves the goal of zero-loss Internet service recovery and upgrade. We make Internet services become fault tolerant in a single node. Our framework can detect the faults and recover the faulty Internet service automatically. It can also reach online maintenance when the Internet service is running in a single node. In addition, we provide some techniques and APIs to enhance FT-TCP which can reduce recovery time in some Internet services. Our framework is divided into two parts - *OS layer Zero-loss Framework(OZS)* and *VMM Zero-loss Framework(VZS)*. They provide some functionalities to reach our goal. They implemented in the kernel and VMM layer. The experimental results show the low overhead in the state logging and acceptable performance during the recovery.

REFERENCES

[1]. 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.

[2]. 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.

[3]. A. B. Brown, D. A. Patterson, "Undo for Operators: Building An Undoable E-mail Store", In proceedings of USENIX Annual Technical Conference, pp. 1-14, Jun. 2003.

[4]. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. "Xen and the Art of Virtualization." In Proceedings of the 19th ACM Symposium on Operating Systems Principles, pages 164-177, October 2003.

[5]. A. Chou, J. Yang, B. Chelf, S. Hallem, Dawson Engler, "An Empirical Study of Operating System Errors", In Proceedings of the 18th ACM symposium on Operating Systems Principles, pp. 73-88, 2001.

[6]. 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.

[7]. 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.

[8]. Y. Chawathe, E. A. Brewer, "System Support for Scalable and Fault Tolerant Internet Service", In proceedings of IFIP International Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware '98), Sep. 1998.

[9]. G. Candea, A. Fox, "Crash-only Software", In proceedings of the 9th Workshop on Hot Topics in Operating Systems, pp. 67-72, Jun. 2003.

[10]. G. Candea, J. Cutler, A. Fox, “Improving Availability with Recursive Microreboots: A Soft-state System Case Study”, Performance Evaluation Journal, Vol. 56, No. 1-3, Mar. 2004.

[11]. C. C. J. Li, W. K. Fuchs, "CATCH-compiler-assisted Techniques for Checkpointing", In proceedings of 20th Annual International Symposium on Fault-Tolerant Computing, pp 74-81, Jun. 1990.

[12]. David E. Lowell, Yasushi Saito, and Eileen J. Samberg, "Devirtualizable Virtual Machines Enabling General, Single-Node, Online Maintenance", In proceedings of the 11th international conference on Architectural support for programming languages and operating systems, pp. 211 – 223, Oct. 2004

[13]. Y. Huang, C. Kintala, N. Kolettis, N. D. Fulton, "Software Rejuvenation: Analysis, Module and Applications", In Proceedings of the 25th International Symposium on Fault Tolerant Computing, pp. 381-390, Jun. 1995.

[14]. HP NonStop Group. Personal communication, 1998.

[15]. David Lorge Parnas, “Software Aging”, In Proceeding of the 16th international conference on Software engineering, pp. 279 – 287, May. 1994

[16]. J. Long, W. K. Fuchs, J. A. Abraham, "Compiler-assisted Static Checkpoint Insertion", In proceedings. of the 22th Annual International Symposium. on Fault-Tolerant Computing, pp. 58-65, Jul. 1992.

[17]. D. Maltz and P. Bhagwat. “TCP Splicing for Application Layer Proxy Performance”, IBM Research Reprot 21139 (Computer Science/Mathematics), IBM Research Division, March 1998.

[18]. 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.

[19]. Performance Technologies Inc.., “The effects of network downtime on profits and

productivity - a white paper analysis on the importance of non-stop networking”, available at http://whitepapers.informationweek.com/detail/RES/991044232_ 762.html, 2001.

[20]. D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, N. Treuhaft, "Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies", Computer Science Technical Report UCB//CSD-02-1175, Mar. 2002.

[21]. C. R. Landau, "The Checkpoint Mechanism in KeyKOS", Proc. Second International Workshop on Object Orientation in Operating Systems, pp. 86-91, Sept. 1992.

[22]. J. S. Plank, “An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance”, Technical Report UTCS -97-372, Jul. 1997.

[23]. F. Sultan, K. Srinivasan, D. Iyer, L. Iftode, "Migratory TCP: Connection Migration for Service Continuity in the Internet", In proceedings of the 22nd International Conf. on Distributed Computing Systems, pp. 469-470, Jul. 2002.

[24]. J. S. Plank, M. Beck, G. Kingsley, K. Li, "Libckpt: Transparent Checkpointing under UNIX", In proceedings of Usenix Winter 1995 Technical Conf., pp. 213-223, Jan. 1995.

[25]. S. T. Hsu, R. C. Chang, "Continuous Checkpointing: Joining the Checkpointing with Virtual Memory Paging", Software Practices and Experiences, vol. 27, no. 9, pp. 1103-1120, 1997.

[26]. 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.