SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2007; **37**:1349–1376
Published online 12 March 2007 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.808

Virtual machine support for zero-loss Internet service recovery and upgrade

Da-Wei Chang[1,*,†], Cheng-En Hsieh[2], Yan-Pai Chen[2] and Kwo-Cheng Chiu[2]

[1]*Department of Computer Science and Information Engineering, National Cheng-Kung University, Tainan, Taiwan*
[2]*Department of Computer Science, National Chiao-Tung University, HsinChu, Taiwan*

SUMMARY

Software is critical for Internet service availability since an Internet service may become unavailable due to software faults or software maintenance. In this paper, we propose a framework to allow zero-loss recovery and online maintenance for Internet services. The framework is based on the virtual machine (VM) technology and a connection migration technique called FT-TCP. It can recover transient application/operating system faults and it allows fault recovery and online maintenance on a single host. The framework substantially enhances FT-TCP so that it can be run efficiently in the VM environment. Specifically, we propose techniques to reduce the inter-VM switches and communication. Moreover, we propose service-specific optimizations to reduce the recovery time of FT-TCP. Finally, the framework provides an interface for the service designers to implement more service-specific optimizations. The framework was implemented by modifying an open source VM monitor, Xen, and the Linux kernel running on top of Xen. The effectiveness and efficiency of the framework were evaluated by running two Internet services, WWW proxy and FTP, on the proposed framework and measuring the impact on their performance. According to the experimental results, our approach causes only slight performance overhead (i.e. less than 4%) and memory overhead (i.e. less than 750 KB) for both the services. Copyright © 2007 John Wiley & Sons, Ltd.

Received 7 December 2005; Revised 30 October 2006; Accepted 19 December 2006

KEY WORDS: service recovery; online maintenance; virtual machine; connection migration

1. INTRODUCTION

In recent years, Internet services have gained great popularity in our daily lives. However, an Internet service may become unavailable due to transient errors, software bugs, and system maintenance.

*Correspondence to: Da-Wei Chang, Department of Computer Science and Information Engineering, National Cheng-Kung University, Tainan, Taiwan.
†E-mail: david.oslab@gmail.com

According to previous research [1], a few minutes of downtime can lead to substantial financial losses. Thus, high availability is an important factor for Internet services.

Previous research [2] has shown that software failures lead to a larger portion of system downtime than hardware faults. Moreover, the latter can be masked by component redundancy [3–5] and therefore software plays a critical role in system availability. Software may crash due to various reasons. As indicated in previous research [6,7], human error is the dominant source. For example, an administrator may misconfigure an Internet service or kill the service unintentionally, causing it to become unavailable. Moreover, transient faults or software aging faults [8,9] may occur on Internet services because of their long execution time. Finally, operating systems (OSs) under the Internet services can also crash since it is difficult to make them error-free due to their high complexity [10]. In addition to failures, software maintenance is another dominant source of downtime for Internet services [11]. For example, an Internet service usually has to stop during system maintenance, which may involve upgrading the service applications, libraries, OS, or drivers. Although the system can be restarted after the maintenance operation has completed, the service state and the OS state (such as the TCP connections) will be lost, which is unacceptable for many commercial services.

Many fault-tolerant techniques have limited ability to solve the above problems. Checkpointing [12] cannot deal with software aging faults, and it usually causes a large overhead. Connection migration techniques [13–17] cannot recover online requests in a server-transparent way. Moreover, they require expensive server replicas. Some connection migration techniques [14–16] even require modifications to the client-side TCP implementations, which limits the feasibility of these techniques. Recursive Recovery [18–20] requires the service to be composed of many fine-grained components, which leads to performance degradation.

In this paper, we propose a framework to achieve the goal of zero-loss Internet service recovery and upgrade on a single host. It allows an Internet service to survive transient software failures (including software aging faults) and system maintenance operations without expensive server replicas and client-side TCP modification. Based on a client-transparent connection recovery technique called FT-TCP [13] and a virtual machine monitor (VMM) called Xen [21], the framework runs the backup server (including the service application and the OS) in another VM (i.e. the backup VM). It can detect application and OS faults, log connection states during the normal operation period, and perform recovery when a fault is detected.

The following features make the framework unique. First, it allows online software maintenance and fault recovery on a single host. Second, it reduces the resource usage of the backup VM and the inter-VM communication during the normal operation period so as to greatly improve the performance of FT-TCP on a VMM. As shown in Section 5.4, simply applying a connection migration technique such as FT-TCP on a VMM with no optimizations results in an unacceptable performance degradation. We eliminate most of the performance degradation to make FT-TCP feasible on a VMM. Third, we propose service-specific optimizations to reduce the recovery time of FT-TCP, and the framework provides an interface for the service designers to implement more service-specific optimizations.

We evaluate the effectiveness and efficiency of the framework by running two Internet service programs, Squid proxy server and Proftpd FTP server, on the proposed framework and measuring their performance impacts. According to the experimental results, our approach incurs little overhead both in terms of performance (i.e. less than 4%) and memory space (i.e. less than 750 KB).

The rest of this paper is organized as follows. Section 2 describes the background technique, FT-TCP. Section 3 presents the design and implementation of our framework. Section 4 shows the possible

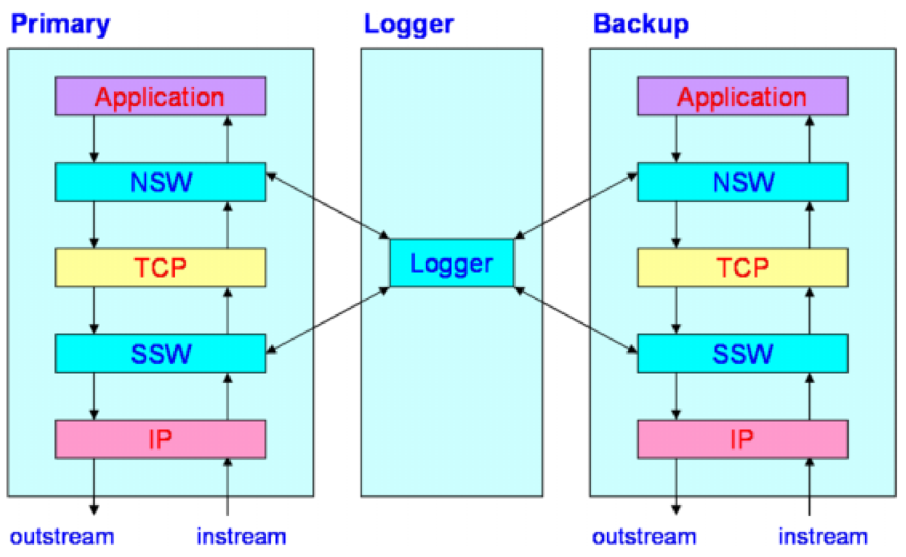

Figure 1. Overview of FT-TCP.

improvement on FT-TCP. Section 5 shows the experimental results, and Section 6 a description of related efforts. Finally, we give discussions and conclusions in Sections 7 and 8, respectively.

2. BACKGROUND

As mentioned in Section 1, our zero-loss service recovery framework is based on FT-TCP. In this section, we introduce the FT-TCP [13] technique and the control flow it uses to recover an Internet service.

2.1. FT-TCP system

FT-TCP involves three logical entities: primary, backup, and logger, where the logger is co-located with the backup. As shown in Figure 1, FT-TCP inserts wrappers around the TCP layer of the primary and the backup machines. The wrappers intercept all the outgoing/incoming TCP packets on the primary machine and record the packet information to the logger machine. If the primary machine fails, FT-TCP recovers the online 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 the TCP. It records the results of specific system calls issued by the service application in order to ensure deterministic execution. For example, it logs the returned length of each socket read operation and the result of each *gettimeofday()* system call. During the recovery period, the NSW returns exactly the same values (which are logged) to the application to maintain determinism[‡]. Suppose that during the normal operation period a service performs two socket read operations and the results, 500 and 1000 bytes, respectively, are logged. During the recovery period, the NSW ensures that the socket read operations issued by the restarted service also return 500 and 1000 bytes, respectively.

The wrapper layer below the TCP is called the south side wrapper (SSW). During the normal operation period, SSW records the client requests of each online TCP connection to the logger. In addition, it also records the number of bytes received from and sent to each of the clients. During the recovery period, the SSW re-establishes the online TCP connections on behalf of the clients according to the logged content to enable the service to continue.

2.2. 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 request-processing flow from the beginning.

For the connection re-establishment, the SSW sends a fake SYN packet to its local TCP (i.e. the TCP of the backup machine). When receiving a SYN packet, the TCP sends a SYN/ACK packet back, which is intercepted and then discarded by the SSW. The SSW then 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 do not need to be adjusted since SSW uses a special initial sequence number, which is equal to $last\_ack\_no - 1$ (where the *last_ack_no* represents the last ACK sequence number from the primary server), in the SYN packet it fakes. Finally, the SSW sends a fake ACK packet to complete the TCP three-way handshake.

After the connection is re-established, the service application replays the request-processing flow (i.e. accept the connection, read the request, process the request, and write the response) with the help of the NSW. For example, when the service application issues a socket read operation, the NSW obtains the logged return length and returns that number of bytes from the logged request. As another example, NSW is also responsible for dropping duplicated response data.

As mentioned above, FT-TCP logs the client requests and the results of specific system calls for all the online connections. To avoid overwhelming the memory, the log can be stored on a disk.

3. DESIGN AND IMPLEMENTATION

In this section, we describe the design and implementation of the proposed framework. We first give an overview of the system components in the framework and then describe the fault recovery and the online maintenance techniques.

[‡]FT-TCP assumes that all the factors that affect deterministic execution can be handled in this way. Thus, a restarted service can have exactly the same behavior as the original.

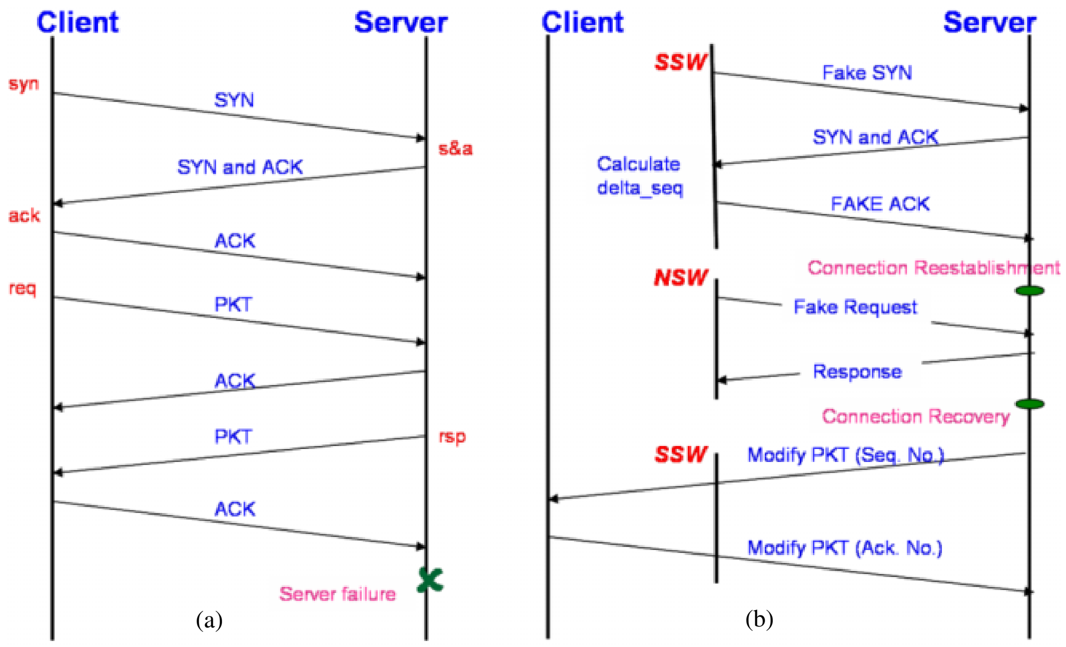

Figure 2. (a) Normal connection setup and (b) connection recovery.

3.1. System components

As shown in Figure 3, we implement our framework in both the OS (i.e. Linux) and the VMM (i.e. Xen). The former part is called the *OS layer zero-loss subsystem* (OZS), while the latter part is called the *VMM layer zero-loss subsystem* (VZS).

The major components of the framework are the protocol manager, health monitor, and recovery manager. The protocol manager is responsible for managing the boot-up/suspension/resumption of the backup VM. The health monitor is used for detecting service application and OS failures. Finally, the recovery manager is responsible for recovering TCP and service states on the backup VM. In addition to the components, we enhanced FT-TCP and provided an interface to allow the service designers to implement their service-specific optimizations on FT-TCP. Moreover, the framework provides system calls for the administrators to control the backup server and the service migration.

3.2. Fault recovery

As mentioned previously, our framework is based on FT-TCP, which involves the primary server, the backup server, and the logger. We ran the primary server on one VM and the backup server on another. Originally, the logger ran in the backup VM or in a dedicated VM. However, this caused frequent

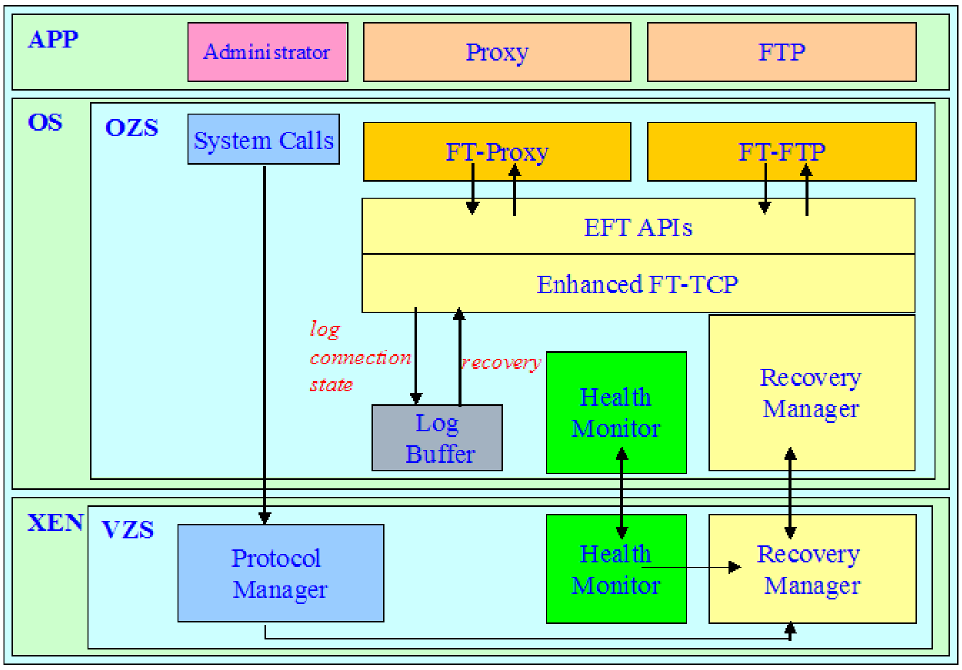

Figure 3. System components.

inter-VM communication (owing to the traffic of connection state logging between the primary VM and the logger). To eliminate such traffic, we provided a persistent log buffer accessible both by the primary and the backup VMs to store the connection state. Moreover, we suspended the backup VM during normal operation so that it does not contend for CPU resources with the primary VM. Once the primary fails, the backup server is resumed to take over the job of the primary. Finally, we provided a fault-detection mechanism for detecting application or OS faults and a recovery mechanism to recover the service state.

Figure 4 shows the recovery flow. Before starting an Internet service, the administrator starts a backup server. Then, in order to supply the primary server with the overall system resources, the backup server releases its CPU time. The primary server then performs the normal operations and logs the connection information. When a fault is detected, the VMM wakes up the backup server and recovers the service state so that the system can provide the service continually. Note that we do not address the issues of recursive failover. Therefore, we assume that no further failures occur during the recovery period.

In the following, we first describe the details of booting-up and suspending the backup server. Then, the connection logging approach is presented, which is followed by a description of the fault-detection mechanism and the recovery flow.

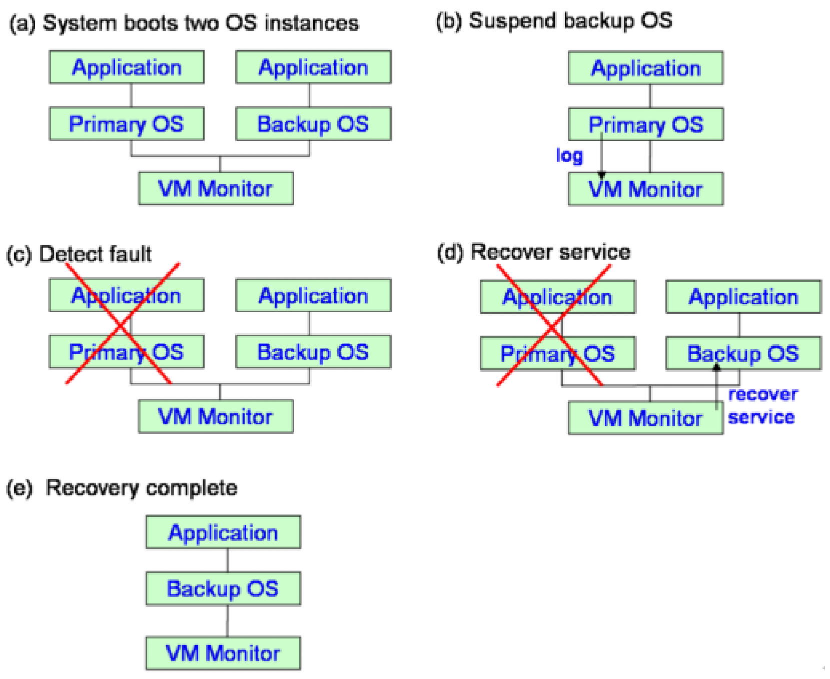

Figure 4. Overview of fault recovery.

3.2.1. Backup server boot-up

To manage the boot-up of the backup server, we developed a protocol involving the VMM and three domains: control[§], primary, and backup. The domains implement the protocol based on the API provided by our framework, as shown in Table I.

Figure 5 shows the flow for booting up a backup server. Originally, Xen only allows the control domain to boot up the 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 have the right to boot up their backups. Specifically, the administrator can register an entry for each primary server that has the right in the *backup-grant* table in advance. The table is stored in VMM and maintained by the protocol manager, and the registration is made by calling the *sys_ins_auth()* system call in the control domain. When a primary server boots a backup server (via the *sys_boot_backup_server()* system call), the protocol manager will check to see if the primary server has the grant.

[§]A domain refers to a VM on the Xen VMM, and the OS in a domain is called the guest OS. Xen has a special domain called the control domain to allow the administrator to manage the other domains.

Table I. System call API provided by the framework.

Function name	Description
sys_ins_auth()	Install authentication information to the VMM
sys_boot_backup_server()	Boot the backup server
sys_change_backup_ip()	Change the backup's IP address
sys_suspend_backup_server()	Suspend the backup server
sys_wakeup_backup_server()	Wake up the backup server
sys_migrate_service()	Notify the recovery manager to migrate the service

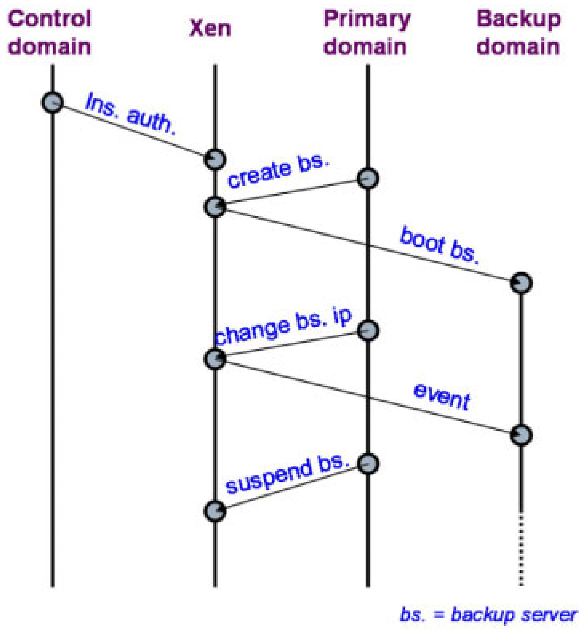

Figure 5. Flow for booting a backup server.

Originally, Xen gives a unique IP address to each guest OS so that each domain can communicate with external hosts. This results in a longer recovery time since the backup server needs 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 receive the primary IP address from the VZS and change its IP address accordingly. The IP address changing is performed 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 VM task from the run queue of Xen. Note that the system call is not required if Xen can automatically suspend an idle VM. However, Xen version 1.2, which is the implementation we use in this paper, does not provide such a capability.

Although the above system calls are implemented in the OZS, most of them require cooperation from the VZS. Communication between the OZS and the VZS is achieved through hypercalls[¶] and events.

3.2.2. *Connection state logging*

As mentioned above, we provide a persistent log buffer to record the connection state of the primary server[‖]. The buffer remains alive even when the primary server crashes. We use a memory area of the primary server as the log buffer. During the recovery period, the backup server remaps the log buffer into its virtual address space and recovers the service state accordingly.

3.2.3. *Fault detection*

Software faults can occur in service applications and OSs. In the following, we describe how to detect the faults.

An application fault is detected by intercepting abnormal process terminations. Specifically, the fault detection is implemented by intercepting the *do_exit()* function in Linux to check its *error_code* parameter. A specific bit of the *error_code* parameter will be turned on if the process termination is caused by a fault (e.g. segmentation violation, trap, etc.). Therefore, we can simply check the bit to see if the process termination is abnormal.

To detect an OS fault, we inserted a *heartbeat generator* in the primary OS and a *heartbeat checker* in Xen. At each timer interrupt, the former sends a heartbeat to Xen by increasing the value of the variable *heartbeat counter* by one, which is shared by the primary domain and Xen. The latter checks the variable at each timer interrupt to detect OS faults. If the value remains the same for the period of two successive timer interrupts (i.e. about 20 ms), the OS is considered to have failed 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 a hypercall, and thus it eliminates the overhead of frequent privilege mode crossings.

3.2.4. *Recovery flow*

When a fault is detected, the recovery manager will follow the recovery protocol for system recovery. Figure 6 illustrates the protocol, which is divided into three steps. First, the recovery manager must

[¶]Similar to system calls that are an interface provided by an OS, hypercalls are an API provided by Xen VMM.

[‖]Since the connection state that we log (i.e. the content of the log buffer) is the same as that logged by FT-TCP, we will skip its description here.

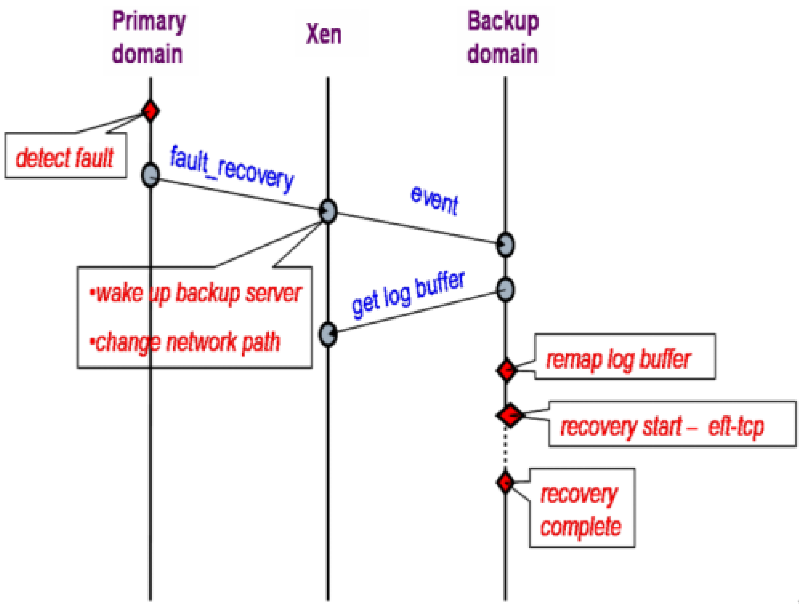

Figure 6. Recovery flow.

change the network path so that incoming packets which were 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 be performed by simply 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 an event to notify the backup server to recover the system. When receiving the event, the recovery manager in the backup server obtains the physical address of the log buffer through a hypercall, remaps the log buffer, and then executes the FT-TCP recovery flow.

It is worth noting that, if the fault does not crash the kernel of the primary domain, the administrator can assign a new IP address to the primary domain and connect to that domain to perform fault diagnosis.

[**]Although the primary and the backup domains share a single IP address, the IP-to-domain mapping maps the IP address to exactly one domain at any given time. Therefore, to determine the target domain, the packet delivery code does nothing more than query the mapping.

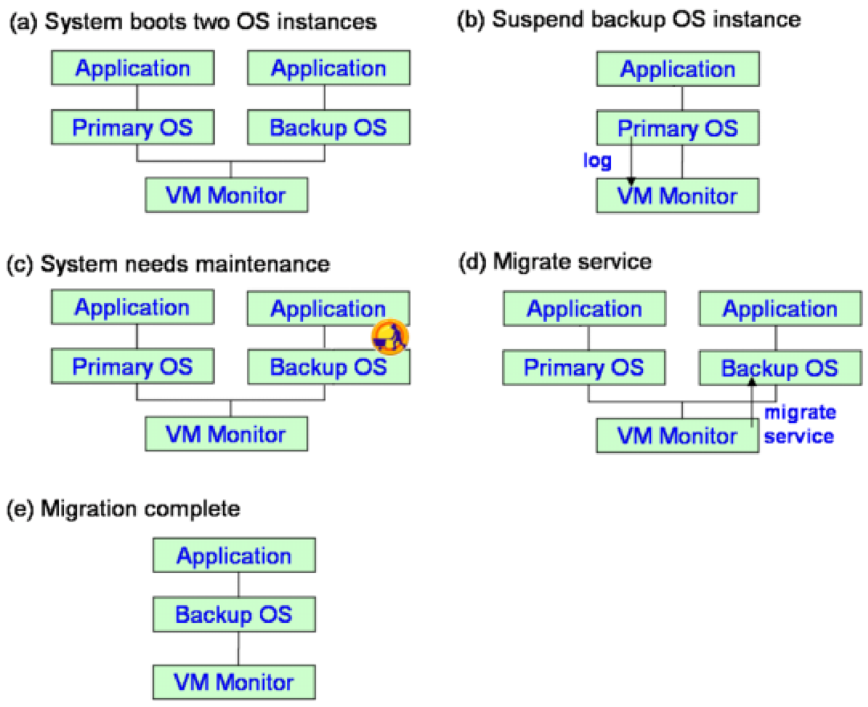

Figure 7. Overview of the online maintenance flow.

3.3. Online maintenance

Most of the mechanisms for online maintenance are the same as those for fault recovery, for example a backup server and a log buffer are also used. The main difference is that we provide a *sys_migrate_service()* system call (as shown in Table I) to allow explicit service migration when the administrator completes system maintenance.

Figure 7 gives a brief overview of the online maintenance flow. First, as shown in Figures 7(a) and 7(b), the protocol manager of the primary VM boots up the backup VM and then suspends it. When the system needs maintenance, the protocol manager wakes up the backup VM so as to allow the administrator to perform system maintenance on it, as shown in Figure 7(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 7(d). Finally, as shown in Figure 7(e), the backup server takes over the job, and the primary server can be suspended.

3.3.1. Maintenance flow

Figure 8 shows the detailed flow for achieving online maintenance. As mentioned in Section 3.2.1, the backup server suspends itself after it has initialized, and after 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.

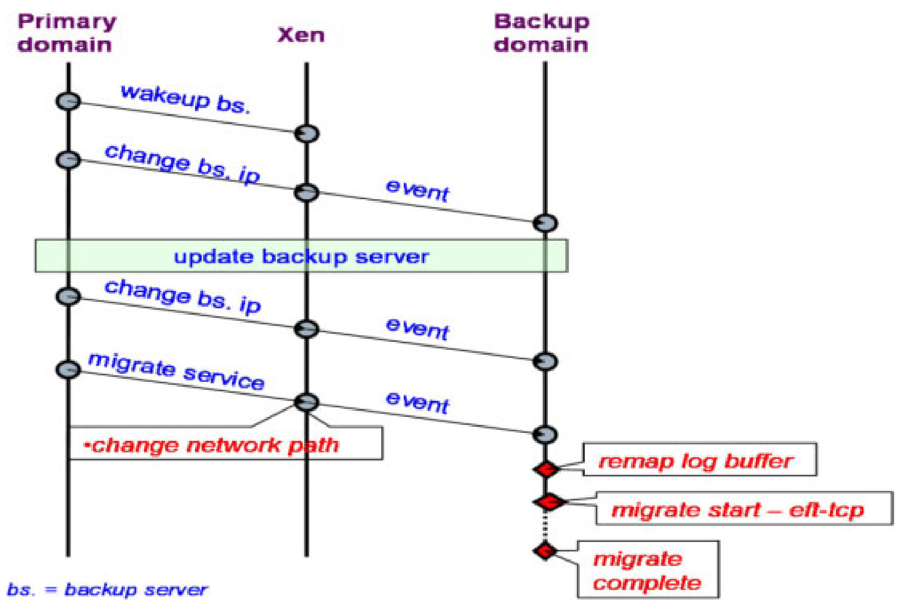

Figure 8. Flow of service migration.

Therefore, to allow online maintenance, the administrator first wakes up the backup server and restores 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 back to that of the primary server and calls the *sys_migrate_service()* system call (as shown in Table I) to explicitly migrate the service. This system call notifies the recovery manager to migrate the service by using the strategy mentioned in Section 3.2.4.

4. RECOVERY TIME REDUCTION

For some Internet services, it may take a long time for FT-TCP to recover them. In this section, we take two Internet services (FTP and HTTP proxy) as examples to show that service-specific optimizations can reduce the recovery time. To ease the implementation effort, the framework provides an interface for Internet service developers to implement such optimizations.

Many Internet protocols, such as FTP, SAMBA, and HTTP 1.1, use long-lived connections/sessions for multiple object transmission. FT-TCP may take a long time when recovering such services since it replays the connection re-establishment and object transmission from the beginning. We use FTP as an example of this recovery process. As shown in Figure 9, 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 replay all the control and data commands.

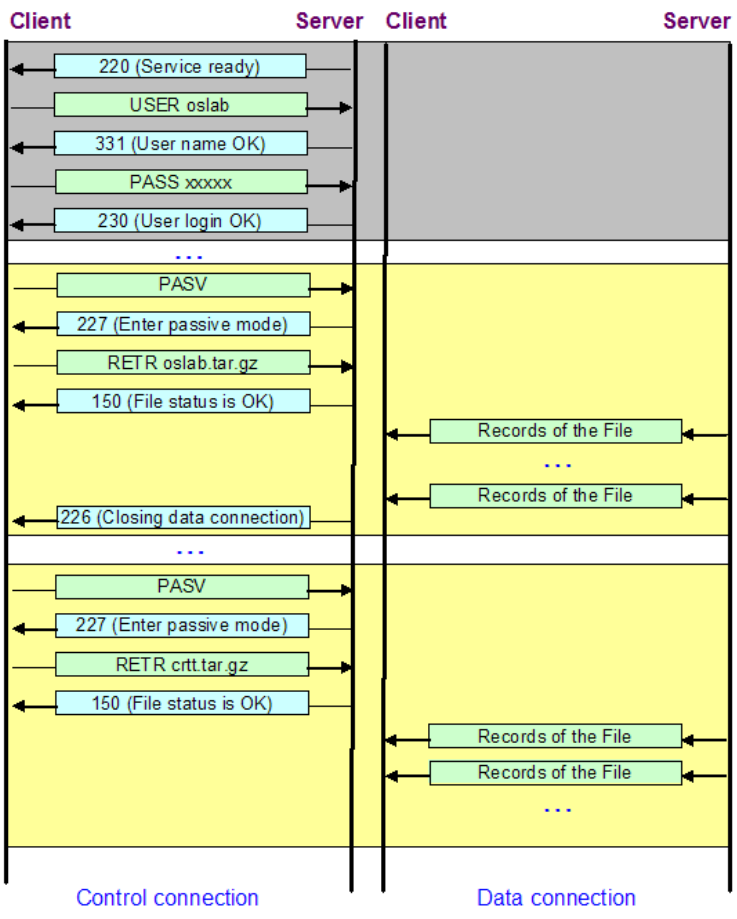

Figure 9. Examples of using FTP for sending files.

However, as shown in the figure, the first set of data commands does not need to be replayed since the corresponding connection has been successfully completed before the fault occurs. Removing such data commands could reduce the recovery time.

A similar situation occurs on relay servers such as proxies[††]. Although FT-TCP can recover a system in a client-transparent way, the recovery is not transparent to end-servers. Specifically, applying FT-TCP on a relay server would cause it to establish a new connection and send the request again to

[††]We refer to servers that are responsible for relaying requests and responses as *relay servers*. Such servers also act as clients. The other servers that act as servers only are called *end-servers*.

Table II. Wrapper registration API.

Function name	Main operations
wr_create()	Create a wrapper structure for the service
wr_ins_nsw()	Install the NSW
wr_ins_ssw()	Install the SSW
wr_unins_nsw()	Uninstall the NSW
wr_unins_ssw()	Uninstall the SSW

the end-server for each request that needs to be recovered. This may cause problems for dynamic-object or transaction-based requests. In addition, it results in a longer recovery time. To achieve end-server transparency and reduce the recovery time, the developer of the fault-tolerant relay service can implement a counterpart of FT-TCP. During the recovery period, the counterpart drops the requests that are resent to the end-server and re-feeds the responses, which have already been received from the end-server, to the relay service application.

Applying such service-specific optimizations on FT-TCP is not easy since FT-TCP itself is a mass of functions and does not provide a clear API for its users. In order to reduce the workload for the developers, we decomposed FT-TCP into several basic operations and export APIs for invoking those operations. Thus, when implementing a fault-tolerant service, the developers can invoke the operations provided by FT-TCP according to their needs. In addition, we have implemented operations to support the above optimizations. For example, a fault-tolerant FTP service can invoke operations to remove requests that no longer need to be replayed during the recovery period. As another example, a fault-tolerant proxy service can invoke an operation to drop the resent requests to the end-server during the recovery period.

Tables II–IV show part of the exported API, which can be divided into three categories. The first category is *wrapper registration* API (shown in Table II), by which application designers can install their wrapper functions. Specifically, designers invoke *wr_ins_nsw()* and *wr_ins_ssw()* to install their NSWs (i.e. above the TCP) and SSWs (i.e. below the TCP), which are invoked whenever a packet goes through the TCP. The second category is the *client-side* API (shown in Table III), which implements the original functionality of FT-TCP and can be used to log information related to client connections. For example, if a SYN packet is received from a client, we can invoke the *cs_rec_syn()* function to create an entry in the log buffer to record information about the new connection. Alternatively, if the application issues a socket read operation, the *cs_read_op()* function can be invoked to log the return value. From the above description we can see that as a general rule for invoking this API we should find out the type of the packet the server currently sends/receives and call the corresponding function. We also provide a *server-side* API, which implements a counterpart of FT-TCP and thus can be used to log information related to server connections. However, we do not describe it in this paper owing to space limitations. The third category is the *application-specific* API (shown in Table IV), which helps service designers to perform service-specific optimizations. Currently, we implemented two functions *as_remove_request()* and *as_remove_response()* for services that utilize persistent connections to remove requests and responses that no longer need to be replayed during the recovery period.

Table III. Client-side API.

Function name	Main operations
cs_rec_syn()	Create a new log entry for the connection
cs_rec_data()	Record the ACK sequence number and packet payload during normal operation; perform packet adjustment during the recovery period
cs_rec_fin()	Prepare to close a connection and free the log entry
cs_rec_rst()	Free the corresponding log entry
cs_snd_ack()	Record the ACK sequence number during normal operation; perform packet adjustment during recovery period
cs_snd_saa()	Calculate *delta_seq* during recovery period
cs_snd_fin()	Prepare to close a connection and free the log entry
cs_snd_rst()	Free the corresponding log entry
cs_read_op()	Record the return value of read during normal operation; provide request data from the log buffer during recovery period
cs_write_op()	Drop duplicated responses sent to the client during recovery period

Table IV. Application-specific API.

Function name	Main operations
as_remove_request()	Remove the specified requests in the log buffer
as_remove_response()	Remove the specified responses in the log buffer

Now we describe how to use the APIs to implement a fault-tolerant FTP system. First of all, the developers should register their wrappers by calling *wr_ins_nsw()* and *wr_ins_ssw()*. In the wrappers, the client-side API should be invoked when necessary. For example, when a data command is received, *cs_rec_data()* can be called to log the command. When a file is completely transferred, the FTP server application sends a specific reply (i.e. reply number 226) to notify the client. Upon intercepting the reply, the SSW can invoke *cs_remove_request()* to remove the commands corresponding to the data connection. When a fault occurs, the logged commands will be replayed with the help of FT-TCP. The commands that were removed will not be replayed during the recovery.

Note that, in our framework, input from the service developers are required only when they want to implement service-specific optimizations, and the service applications do not need to be modified.

5. PERFORMANCE EVALUATION

In this section, we evaluate the effectiveness and efficiency of our framework. First, we measure the overhead of our framework and then measure the recovery time. Finally, we demonstrate the effectiveness of the online maintenance support of our framework.

Table V. Experimental environment.

	Client	FT machine	End-server
Hardware	P4 1.6 GHz, 256 MB memory	P4 2.0 GHz, 1 GB memory	P4 2.0 GHz, 256 MB memory
VMM	N/A	Xen 1.2	N/A
OS	Linux 2.4.18	Xenolinux 2.4.26	Linux 2.4.18
Guest domains	N/A	Control domain—64MB Primary domain—256 MB Backup domain—256 MB	N/A
Software	http_load 12mar2006, dkftpbench 0.45, wget	Squid-2.5.STABLE4, Proftpd-1.2.8	Apache 2.0.40

5.1. Experimental environment

As shown in Table V, we run the experiments using three machines, one for clients, one for the fault-tolerant system (i.e. the FT machine), and the other for the end-server, all three of which are connected via 100 Mbps fast Ethernet links. The client machine is a Pentium 4 1.6 GHz PC with 256 MB of memory, running Linux kernel 2.4.18 and benchmark applications such as http_load version 12mar2006 [22], and dkftpbench version 0.45 [23]. The end-server machine is a Pentium 4 2.0 GHz PC with 256 MB DRAM, running Linux kernel 2.4.18 and Apache server version 2.0.40. The FT machine is an Intel Pentium 4 2.0 GHz PC with 1 GB DRAM, running Xen 1.2 as the VMM, Xenolinux 2.4.26 as the guest OS, and Squid-2.5.STABLE4 [24] and Proftpd-1.2.8 [25] as the applications. We allocate 64 MB, 256 MB, and 256 MB of memory for the control domain, primary domain, and backup domain, respectively.

5.2. Performance overhead

In this experiment, we use the http_load benchmark to measure the impact of state logging on the performance of Squid. The benchmark runs a number of concurrent clients, each of which requests files from the Web server via the Squid server. The size distribution of the requested files is obtained from the Webstone benchmark [26]. Figure 10 shows the comparison of the client-perceived response time with and without our framework. The white bars indicate the response time running on the normal OS and the black bars indicate the response time running on our framework. From this figure we can see that using our framework results in only slight performance degradation, ranging from 1 to 4%.

In addition to the performance overhead on Squid, we also measure the performance overhead of our framework on a FTP service program, Proftpd, by using the dkftpbench benchmark version 0.45. Figure 11 shows the performance scores of Proftpd with and without our framework. The scores are reported by dkftpbench, and the figure shows that the throughput degradation is only about 1.12%. These figures demonstrate that our framework has little impact during normal operation.

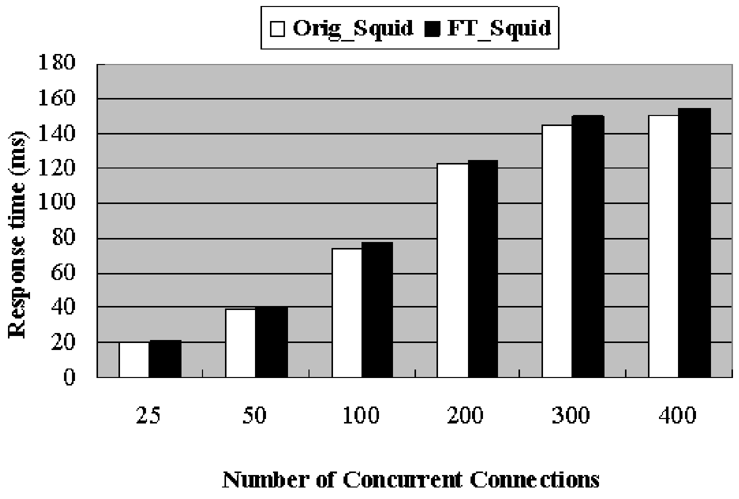

Figure 10. Performance comparison on Squid.

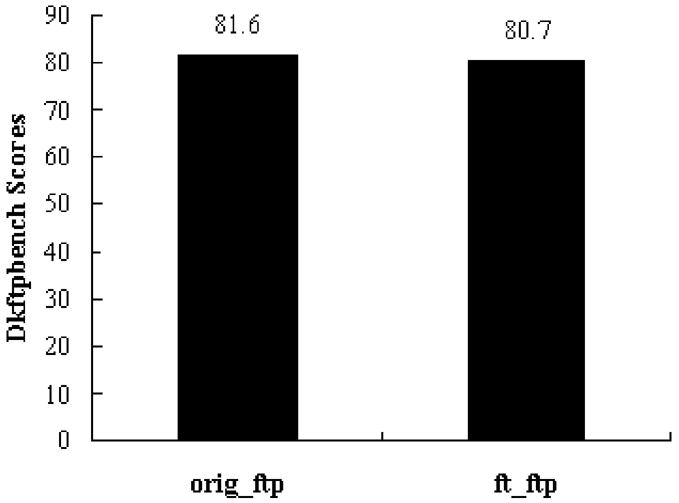

Figure 11. Performance of Proftpd (connection throughput).

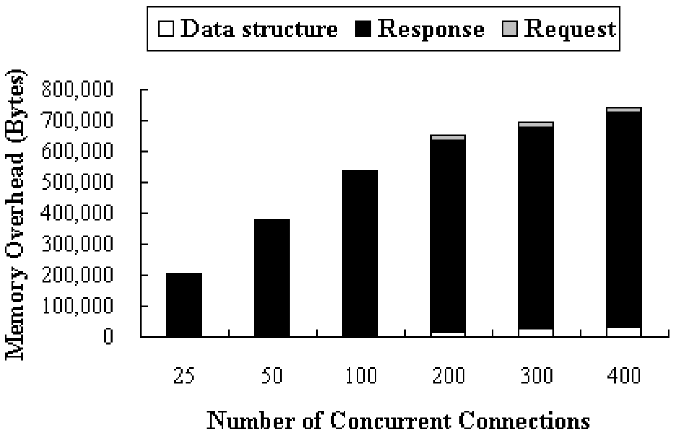

Figure 12. Memory overhead when running Squid.

5.3. Memory overheads

In this section, we measure the memory overhead (i.e. the size of the in-memory log buffer) of our framework. Typically, the overhead comprises three parts: logged requests, logged responses, and data structures for maintaining connection logging. In the first experiment, we use the http_load benchmark to make a number of concurrent requests to the Squid proxy server and measure the maximum size of the log buffer. The size distribution of the requested files is obtained from the Webstone benchmark (see Figure 12 for the results). As shown in the figure, the overhead is less than 750 KB for 400 concurrent connections, which does not cause much pressure on a server system. Moreover, the logged response data dominates the overhead because the size of a Web response is usually much lager than that of a Web request.

In the second experiment, we measure the memory overhead when running the Proftpd server. The clients are a mixture of wget (60%) and ftp-get (40%) programs, while the latter is implemented by us. Each client downloads a 5 MB file from the server, and each of the ftp-get clients performs a random number of directory changes before downloading the file. As shown in Figure 13, the memory overhead is only about 100 KB for 400 concurrent clients. Moreover, the response data are not logged since they can be regenerated from the file system.

According to the experimental results presented in this section, the in-memory log buffer does not put much pressure on a server system in normal cases. However, there are some cases where the overhead can overwhelm the system memory, which will be discussed in Section 7.

5.4. Effects of reducing inter-VM traffic and switching

In order to demonstrate the performance benefit of reducing the inter-VM traffic, we compare the performance of our framework with that of the original FT-TCP architecture, in which the primary VM

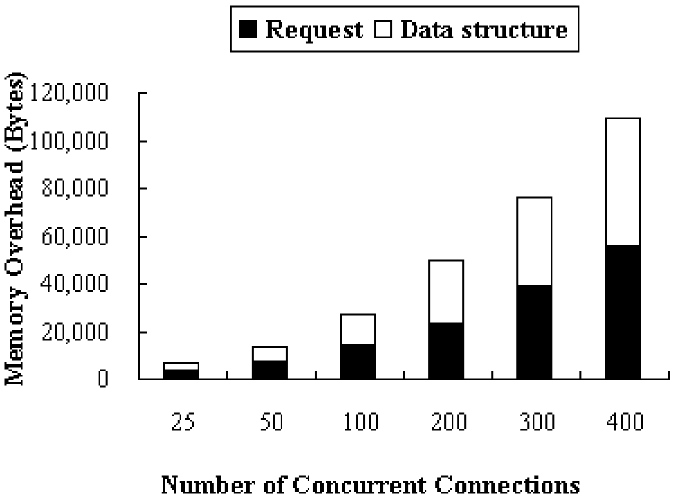

Figure 13. Memory overhead when running Proftpd.

sends a log to the logger VM (i.e. the backup VM). In this experiment, the client machine uses the wget utility to download a 5 MB file from the end-server, via the Squid server run on the FT machine (see Figure 14 for the results). The ORIG, EFT-TCP, and FT-TCP bars represent the performance of the original Squid, the Squid running on our framework, and the Squid running on the original FT-TCP architecture, respectively. As shown in the figure, use of FT-TCP results in a 51.35% performance degradation, which results mainly from the inter-VM switching and traffic. In contrast, our framework does not have this overhead and there is only 1.25% performance degradation, which makes its application on a VMM more feasible.

5.5. Recovery time

In this experiment, we measure the performance of Squid on our framework with the presence of failures. In the experiment, the client requests one file from the server machine through the Squid on the FT machine, and the Squid process is terminated when the first half of the file data is sent to the client. Under this situation, our framework recovers the Squid on the backup domain. For comparison, we also measure the performance of restarting the Squid process on the same domain (i.e. the primary domain). The results are given in Figure 15. The white bars represent the transmission time when no fault occurs, the gray bars represent the transmission time when the fault occurs and is recovered by our framework, and the black bars represent the transmission time when the a fault occurs and is recovered by restarting the Squid server on the same domain. According to the results, recovery in a single domain and in different domains lead to delays of about 600 ms and 300 ms, respectively. Thus, recovery in different domains is more efficient. The reason is that recovery in the same domain

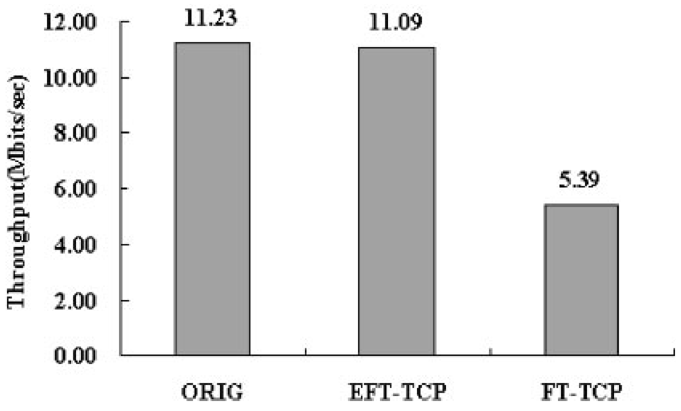

Figure 14. Benefit of reducing the inter-VM traffic and switching.

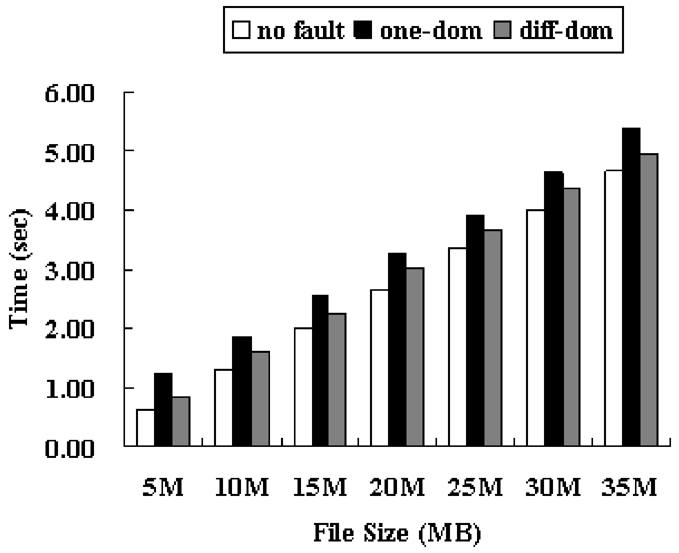

Figure 15. Fault occurring when the first half of the data is sent.

must wait for the re-initialization of the Squid server, which takes about 300 ms in our environment. In contrast, Squid has been initialized and suspended on the backup domain under our framework.

Note that the 300 ms delay is not the real recovery time. Figure 16 shows the measured recovery time of the 35 MB file transfer. The recovery time is divided into three parts. The preparation time

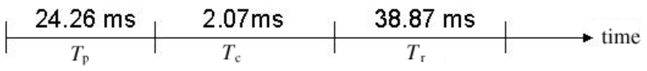

Figure 16. Recovery time of a single connection.

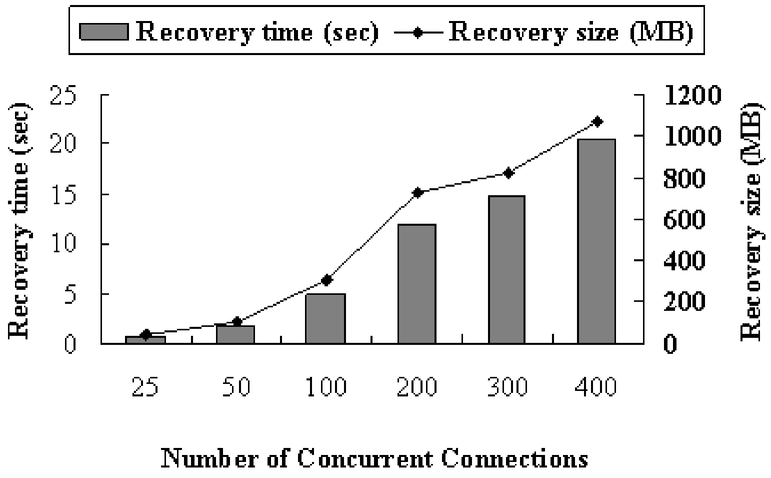

Figure 17. Performance of recovering multiple connections.

T_p represents the time between the occurrence of the fault and the execution of the backup domain. The connection re-establishment time T_c represents the time for re-establishing the connection, and the data recovery time T_r represents the time for replaying the data transfer until the time when the fault occurs. As shown in the figure, the total recovery time is about 65 ms. The extra 235 ms delay comes from the drop of the TCP congestion window size since the connection is closed when the fault occurs and is established again during the recovery period.

Figure 17 shows the time for recovering multiple connections. In this experiment, we use multiple concurrent clients to download files from the Proftpd run on the FT machine. Each client sends six control command requests (including USER, PASS, SYST, PWD, TYPE I, and CWD) and two data command requests (including PASV and RETR) to download a 5 MB file. We inject a fault when the last client has received 2.5 MB of data. As shown in the figure, the size of the data needing to be replayed (i.e. the replay size) grows, which leads to the increment of the recovery time, as there are more clients. To recover 400 pending connections, the system replays about 1 GB of data and takes about 20 s. Note that the 1 GB of data is not stored in the log buffer. Instead, it is reloaded by the FTP server program from the file system. Moreover, the large replay size is the result of the limitation of FT-TCP, which replays each request from the beginning (although we can remove requests that have been done in the FTP case) in order to maintain server application transparency.

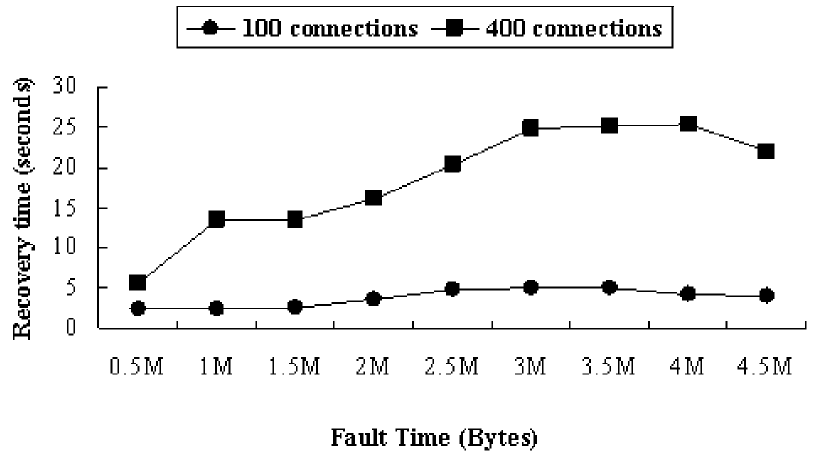

Figure 18. Recovery performance with varying fault time.

Figure 18 shows the recovery time of 100 and 400 pending client connections with varying fault times. We inject faults when the last client has received R bytes of data. As shown in the figure, the recovery time grows as R increases from 0.5 to 3.5 MB as a result of the increment of the recovery size. However, as R increases further, some clients are finished and hence the recovery size decreases, causing the recovery time to drop.

5.6. Online maintenance

In the last experiment, we demonstrate the effectiveness of online maintenance on Proftpd. We download a 5 MB file from a rate-limited Proftpd on the primary domain and manually migrate the service from the primary domain to the backup domain (which contains a rate-unlimited Proftpd) during the file transfer. The transmission rate of the Proftpd on the primary domain is limited to 1 MB s^{-1}, and we measure the average throughput before and after the service migration. Figure 19 shows the results that service migration takes approximately 80 ms and the average throughput is increased to 11 MB s^{-1} after the service migration. This demonstrates that the administrator can upgrade the service on the backup domain and then perform service migration without stopping the service.

6. RELATED WORK

In this section, we first describe related research in transport-layer fault tolerance, and then describe efforts in online maintenance. Finally, other related efforts are mentioned.

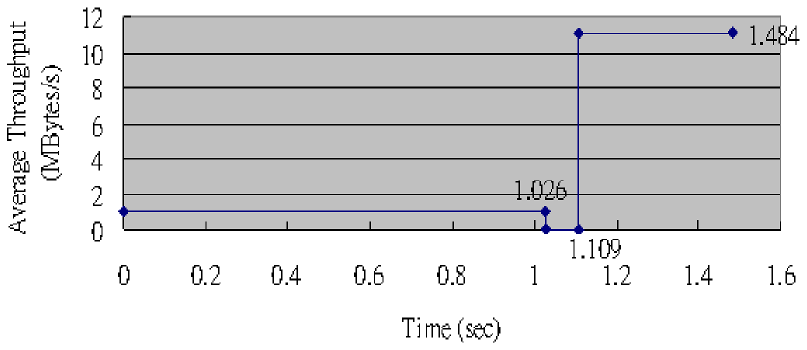

Figure 19. Online maintenance of Proftpd.

6.1. Transport-layer fault tolerance

Snoeren *et al.* [14] proposed a cluster-based approach to improve the availability of a Web service. The approach inserts a HTTP-aware module between the application and the transport layers to log the inter-layer interactions. Moreover, it uses TCP migrate options to record the 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 uses require modification to the TCP implementations of both the server and the clients, reducing the feasibility of the approach.

Migratory TCP (M-TCP) [16] allows online connections to be migrated from one server to another cooperative server. When a server overloads or fails, the migration process will be triggered, which causes the client to reconnect to a server replica with better performance. A set of APIs are provided for the server application to transfer service states between server replicas. However, as in Snoeren *et al.* [14], M-TCP requires extending both the client-side and the server-side TCP implementations to accomplish dynamic connection migration.

TCP Splicing [27] splices two *physical* connections, a client-proxy connection and a proxy-server connection, into a *virtual* client-server TCP connection. As shown in Figure 20, all the clients connect to the 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 migrates the client request to another server in a way that is transparent to both the client and the server. However, this approach requires multiple server replicas and an extra proxy machine. In contrast, we address software failure and perform fault recovery on a single node.

FT-TCP [13] is a client-transparent connection recovery technique. As described in Section 2, FT-TCP 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 occurs. Our approach improves FT-TCP in two ways.

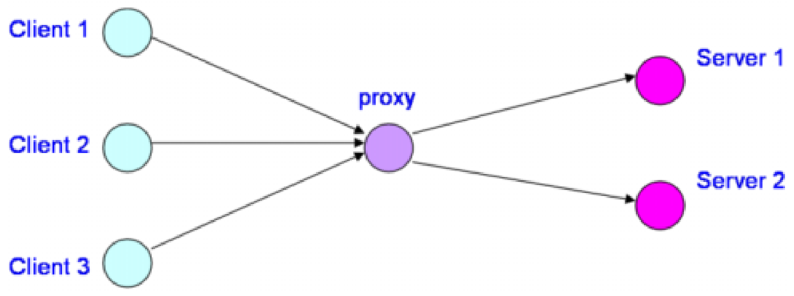

Figure 20. TCP splicing.

First, we make it more feasible to apply FT-TCP on a VMM by reducing the inter-VM traffic and switches. Second, we propose two optimizations for reducing the recovery time of FT-TCP and allow the service designers to implement more service-specific optimizations.

6.2. On-line maintenance

The Cluster Rolling Upgrade [28] provides a way to perform online maintenance by using spare nodes or cluster nodes. In this approach, the administrator upgrades the system on a cluster node (or a spare node) and then migrates the application to that node. However, this approach does not address the problem of online connection migration, and it requires at least one extra spare or cluster node for software maintenance.

Devirtualization [29] enables online maintenance on a single node. It starts the VMM and the backup domain dynamically when a software maintenance operation is needed. Once the maintenance operation is finished on the backup domain the service state can be migrated to that domain, and then the original domain and the VMM can be shut down. This approach provides a good level of performance during normal operations since the VMM is presented only when maintenance is needed. However, it focuses only on decreasing the planned downtime and cannot deal with software faults.

VM Migration [30,31] allows migrating a live VM (including the OS and the application services) from one physical host to another, and thus enables load balancing and low-level maintenance. Since the migration operation involves transmission of the whole VM state, the two main issues are (1) how to perform the background transmission of the memory pages without largely degrading the system performance, and (2) when to stop the source VM and transmit all the remaining pages to the destination. The authors did not address the issues of fault management. In contrast, our work deals with application/OS faults occurring on a VM.

6.3. Other approaches

Autonomic computing [32] proposes self-healing techniques that can automatically detect, diagnose, and repair software and hardware problems. Recovery-oriented computing [2,33] also proposed new techniques to deal with hardware faults, software bugs, and operator errors. The basic principle of these two projects is similar to ours, i.e. systems should deal with faults instead of preventing them.

Checkpointing [12, 34–38] 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 cannot solve the software aging problem since the checkpoint state ages rather than being refreshed, so even if the system can be recovered the software may fail again immediately. Second, checkpointing usually results in a large performance overhead due to the large volume of states that need to be stored.

Recursive Restart (RR) [18,19] and Scalable Network Services (SNS) [20] both allow a fine-grained component-based service to restart a component or a set of components (instead of the whole service) once the component fails, and thus reduce the service restart time. However, the inter-component communication degrades the system performance, which is not suitable for performance-critical Internet services. Moreover, they require an Internet service to be composed of fine-grained components, which requires redesigning the legacy Internet service programs.

Backdoors system [39] can recover a service session state in a cluster environment. When a service node fails, the session state on the failed node is transferred to a backup node via the remote direct memory access (RDMA) mechanism. Similar to our work, the system detects OS failures. However, it requires service applications to checkpoint their state periodically, which requires moderate application modifications. Moreover, it requires special/programmable network cards with a RDMA capability.

Zap [40] allows a group of processes with online connections to be migrated to a new host by using the following two mechanisms. First, it checkpoints the connection state right before the migration so that the state can be restored on the new host. Second, it utilized an end-to-end VNAT (Virtual Network Address Translation) [41] approach and a proxy to direct the traffic to the new host. In contrast with our work, Zap does not address application and OS failures. Moreover, an extra proxy is required for client connection migration.

SSM [42] is a highly-available storage system for the user session state. In the system, stateless application servers access the session state via connection with the storage nodes. SSM improves the availability of the session state by distributing the state to multiple storage nodes. Moreover, each storage node can be restarted and recovered independently if faults have occurred on that node. In contrast with our work, SSM assumes that application servers are stateless, and requires the application services to access the session state on the storage nodes. In contrast, our work does not have such a requirement and is able to recover stateful service applications.

7. DISCUSSIONS

In this section, we provide discussions about the memory overhead and the fault-detection techniques.

As shown by the experimental results, the log buffer does not cause memory pressure on the system in normal cases. However, there exist some pathological cases in which the buffer may overwhelm the system. For example, uploading a huge file to our fault-tolerant FTP server causes memory pressure on the server as a result of the logging of the client request (i.e. the content of the uploaded file). For another example, if the consumption rate of the response data is far behind the production rate in a Web proxy system (e.g. maybe owing to the congestion of the network path between the proxy and the client), the not-yet-consumed response data will cause memory pressure to the proxy system.

These problems can be solved by storing the log buffer into a disk, or by closing the TCP receive window of the server connection temporarily. In the future, we will incorporate these approaches into our framework.

As described previously, we use techniques to detect application and OS faults. However, there are still a number of faults that cannot be detected by these techniques. For example, an infinite loop in a service application cannot be detected since it does not cause the abnormal termination of the application. We can detect such faults by sending test requests to the application periodically and checking the results. Some other OS faults can be detected by catching kernel exceptions, or by tracking the numbers of served interrupts and context switches, as proposed by Sultan *et al.* [39]. In the future, we will incorporate more fault-detection techniques into our framework.

8. CONCLUSIONS

In this paper, we proposed a framework for providing a zero-loss Internet service recovery and upgrade on a single host. Based on Xen and FT-TCP, the framework can recover transient application and OS faults. In addition, it allows online service upgrades without stopping the service. In order to make the application of FT-TCP to VMMs more feasible, we proposed techniques to reduce the inter-VM switches and communication. Moreover, we proposed service-specific optimizations to reduce the recovery time of FT-TCP. Finally, the framework was shown to provide an interface for service designers to implement more service-specific optimizations. We evaluated the effectiveness and efficiency of our framework on two popular service programs, Squid and Proftpd. According to the experimental results, our approach incurs little performance overhead (i.e. ranging from 1 to 4%) and memory overhead (i.e. less than 750 KB).

In the current implementation, the framework integrates optimizations for FTP and proxy services. In the future we will investigate more possible optimizations for other services and integrate them into our framework. In addition, we will provide a better interface for the service designers so that they can *specify* their optimizations, instead of implementing the optimizations by themselves.

REFERENCES

1. Performance Technologies Inc. The Effects of Network Downtime on Profits and Productivity—a White Paper Analysis on the Importance of Non-stop Networking. http://searchtechtarget.techtarget.com/0,293857,sid7_gci827871,00.html [January 2007].
2. Patterson D *et al.* Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. *Computer Science Technical Report UCB//CSD-02-1175*, UC Berkeley, March 2002.
3. Intel Corporation. Intel Networking Technology—Load Balancing. http://www.intel.com/network/connectivity/resources/technologies/load_balancing.htm [2003].
4. Jann J, Browning LM, Burugula RS. Dynamic reconfiguration: Basic building blocks for autonomic computing on IBM pSeries servers. *IBM Systems Journal* 2003; **42**(1):29–37.
5. Patterson D, Chen P, Gibson G, Katz RH. Introduction to redundant arrays of inexpensive disks (RAID). *Digest of Papers for 34th IEEE Computer Society International Conference (COMPCON Spring '89)*. IEEE Computer Society Press: Los Alamitos, CA, 1989; 112–117.
6. Brown A, Patterson DA. To err is human. *Proceedings of the 2001 Workshop on Evaluating and Architecting System Dependability*, Sweden, July 2001.
7. Oppenheimer D, Ganapathi A, Patterson DA. Why do Internet services fail, and what can be done about it? *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Seattle, WA, March 2003.

8. Huang Y, Kintala C, Kolettis N, Fulton ND. Software rejuvenation: Analysis, module and applications. *Proceedings of the 25th International Symposium on Fault Tolerant Computing*, Pasadena, CA, June 1995; 381–390.
9. Parnas DL. Software aging. *Proceeding of the 16th International Conference on Software Engineering*, Sorrento, Italy, 1994; 279–287.
10. Chou A, Yang J, Chelf B, Hallem S, Engler D. An empirical study of operating system errors. *Proceedings of the 18th ACM Symposium on Operating Systems Principles*. ACM Press: New York, 2001; 73–88.
11. HP NonStop Group. Personal communication, 1998.
12. Plank JS. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. *Technical Report UTCS-97-372*, University of Tennessee, July 1997.
13. Alvisi L, Bressoud TC, El-Khashab A, Marzullo K, Zagorodnov D. Wrapping server-side TCP to mask connection failures. *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '01)*, Anchorage, AK, April 2001; 329–337.
14. Snoeren AC, Andersen DG, Balakrishnan H. Fine-grained failover using connection migration. *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS '01)*, San Francisco, CA, March 2001.
15. Snoeren AC, Balakrishnan H. An end-to-end approach to host mobility. *Proceedings of the 6th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, Boston, MA, August 2000; 155–166.
16. Sultan F, Srinivasan K, Iyer D, Iftode L. Migratory TCP: Connection migration for service continuity in the Internet. *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, Washington, DC, July 2002; 469–470.
17. Zagorodnov D, Marzullo K, Alvisi L, Bressoud TC. Engineering fault-tolerant TCP/IP servers using FT-TCP. *Proceedings of IEEE International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society Press: Los Alamitos, CA, 2003; 22–26.
18. Candea G, Cutler J, Fox A. Improving availability with recursive microreboots: A soft-state system case study. *Performance Evaluation Journal* 2004; **56**(1–4):213–248.
19. Candea G, Fox A. Crash-only software. *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, Lihue, Hawaii, June 2003; 67–72.
20. Chawathe Y, Brewer EA. System support for scalable and fault tolerant Internet service. *Proceedings of the 1998 IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, The Lake District, U.K., September 1998.
21. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the art of virtualization. *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003. ACM Press: New York, 2003; 164–177.
22. ACME Laboratories. http_load—Multiprocessing HTTP Test Client. http://www.acme.com/software/http_load/ [October 2006].
23. Kegel D. The dkftpbench Benchmark. http://www.kegel.com/dkftpbench/ [January 2007].
24. Wessels D. Squid Web Proxy Cache. http://www.squid-cache.org/ [January 2007].
25. Morrissey J, Renner M, Roesen D, Saunders TJ. The Proftpd Software. http://www.proftpd.org/ [January 2007].
26. Mindcraft Inc. WebStone: The Benchmark for Web Servers. http://www.mindcraft.com/benchmarks/webstone/ [January 2007].
27. Maltz D, Bhagwat P. TCP splicing for application layer proxy performance. *IBM Research Report 21139*, Computer Science/Mathematics, IBM Research Division, March 1998.
28. Microsoft Corporation. Windows 2000 clustering: Performing a rolling upgrade. *Windows 2000 Technical Resources*, 2000. Available at: http://www.microsoft.com/technet/prodtechnol/windows2000serv/deploy/rollout/rollupgr.mspx [January 2007].
29. Lowell DE, Saito Y, Samberg EJ. Devirtualizable virtual machines enabling general, single-node, online maintenance. *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2004)*, Boston, MA, October 2004; 211–223.
30. Clark C, Fraser K, Hand S, Hansen JG, Jul E, Limpach C, Pratt I, Warfield A. Live migration of virtual machines. *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSD '05)*, Boston, MA, May 2005.
31. Nelson M, Lim BH, Hutchins G. Fast transparent migration for virtual machines. *Proceedings of USENIX 2005 Annual Technical Conference (USENIX '05)*, Marriott Anaheim, CA, April 2005; 391–394.
32. Kephart JO, Chess DM. The vision of autonomic computing. *Computer Journal* 2003; **36**(1):41–50.
33. Brown A, Patterson DA. Undo for operators: Building an undoable e-mail store. *Proceedings of USENIX Annual Technical Conference*, June 2003; 1–14.
34. Hsu ST, Chang RC. Continuous checkpointing: Joining the checkpointing with virtual memory paging. *Software: Practice and Experience* 1997; **27**(9):1103–1120.

35. Landau CR. The checkpoint mechanism in KeyKOS. *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, Dourdan, France, September 1992; 86–91.
36. Li CCJ, Fuchs WK. CATCH-compiler-assisted techniques for checkpointing. *Proceedings of the 20th Annual International Symposium on Fault-Tolerant Computing*, June 1990; 74–81.
37. Long J, Fuchs WK, Abraham JA. Compiler-assisted static checkpoint insertion. *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, Boston, MA, July 1992; 58–65.
38. Plank JS, Beck M, Kingsley G, Li K. Libckpt: Transparent checkpointing under UNIX. *Proceedings of USENIX Winter 1995 Technical Conference*, New Orleans, LA, January 1995; 213–223.
39. Sultan F, Bohra A, Smaldone S, Pan Y, Gallard P, Neamtiu I, Iftode L. Recovering Internet service sessions from operating system failures. *IEEE Internet Computing* 2005; **9**(2):17–27.
40. Osman S, Subhraveti D, Su G, Nieh J. The design and implementation of Zap: A system for migrating computing environments. *Proceedings of 5th USENIX/ACM Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, December 2002; 361–376.
41. Su G, Nieh J. Mobile communication with virtual network address translation. *Technical Report CUCS-003-02*, Columbia University, February 2002.
42. Ling BC, Kcman E, Fox A. Session state: Beyond soft state. *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, 2004; 295–308.