無資料遺失之重新啟動架構在網際網路服務上之設計與實作

研究生：蔡眷民　　指導教授：張瑞川教授

國立交通大學資訊科學系

論　文　摘　要

近年來網際網路上的各種服務愈來愈被廣泛的應用來解決問題或尋找資料。這也使得這些網際網路服務更容易因為長期的運作，或是同時服務大量的客戶，而發生突發性的暫時錯誤。因此，我們提出了一個無資料遺失的重新啟動架構，使得網際網路服務能夠永續的運作。

以往已有一些無資料遺失的重新啟動方法，例如檢查點的建立技術，但是這會造成不小的網際網路服務運作效能減低，而且也並不完全適用於解決突發性的暫時錯誤。另外，FT-TCP 也一個是使網際網路服務能不中斷的方法，但是再重新啟動的時候可能會耗費較多的時間。而我們的架構能夠達到一樣的效果但是又有使得整體效能降低不多，而且重新啟動的時候也不會耗費太多的時間。

我們的架構包含了兩個部分。首先，需要修改網際網路服務使他的運作狀態能被存放在安全的地方，這是為了避免重新啟動之後的資料流失。另外，我們還提供了一些系統核心上的支援，使得網際網路服務所用到的輸入輸出頻道能被保留並且自動的完成整個重新啟動的流程。最後，我們將這個架構套用在一個熱門的網頁伺服器上，並且驗證了我們的想法。

A Framework for Zero-Loss Internet Service Restart through Application-Kernel Cooperation

Student：Chuan-Ming Tsai Advisor：Prof. Ruei-Chuan Chang

Institute of Computer and Information Science

National Chiao-Tung University

Abstract

Internet services are more and more widely used for resolving problems or finding information recently. Thus, Internet services have to run for a long time and must serve a huge number of clients concurrently. This causes them easy to suffer from the transient fault and software aging problem, which will cause serious damage when them occurs at some critical Internet services such as commercial web sites. In order to keep Internet service permanently running, we propose a zero-loss restart framework to resolve the transient fault or aging problem of Internet services.

Our framework consists of three parts, the service state abstraction, the kernel I/O channel keeping kernel support, the automatic restarting mechanism kernel support. In addition, we deploy our framework on a popular tiny web server, *thttpd*.

The former zero-loss restart mechanism such as checkpoint and restart has the shortcoming of high runtime overhead and can not totally resolve the transient fault. Another popular zero-loss restart mechanism is the FT-TCP which also has the shortcoming of high recovery time. As result of experimental, our framework only costs lower than 6.8% runtime overhead and lower than 8.5% recovery overhead to perform the zero-loss restart.

Acknowledgements

I am so grateful to have much guidance from my advisor Professor R. C. Chang. He taught me the essential of research, guided me the way of thinking, and brought the knowledge of operating system to me. I also very appreciate Dr. Da-Wei Chang who advised me so much so that I am able to finish my thesis.

Besides, each member of Computer System Lab. brought me laugh and knowledgeable discussion which pushed me forward. Finally, I would like to thank my parents and my friends. They showed me the unlimited love when I really need it.

Chuan-Ming Tsai
Institute of Computer and Information Science
National Chiao-Tung University
2004/6

CONTENTS

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

1.1. Motivation

In the recent years, Internet services have become more and more popular in our regular life. Moreover, the emergence of many transaction based Internet services such as on-line banking, trading, and shopping has put a requirement of high availability on such services. According to the previous research [18], a few minutes of downtime will lead to a great loss of money for these services.

However, due to the long serving time and huge number of clients, Internet services are easily to suffer from transient faults or software aging problem [10]. The former are caused by transient hardware or human errors, while the latter are mainly due to software bugs[1].

Current approach to workaround the faults is to restart the service. However, the original service state will be lost after the service restart. For example, restarting a web server will result in the lost of all the on-line request information at the application level and all the open file information, including the TCP connection information, at the kernel level. This is unacceptable for many commercial or transaction based web sites.

Previous approaches have limitations for solving this problem. Some can not

[1] Even with serious testing, it is not possible to remove all the bugs in a production system. With the aging of the system, some bugs may reveal and cause the system to crash. This is called software aging problem. And, we call the resulting faults as software aging faults. Strictly speaking, software aging faults are a kind of transient faults. However, we separate the two because they cannot always be resolved by the same approach. The detail will be mentioned later in the related work section.

recover from the software aging faults [19]. Others require a long recovery time or expensive server replicas. Still others put a requirement that a service should be made up of many fine-grained and loosely-couple components, which not only needs large modifications to many existing Internet service programs but also degrades the system performance.

In this thesis, we propose a framework that can achieve the goal of zero-loss service restart. By applying the framework to an Internet service, the problem caused by transient faults and software aging can be resolved. In addition, the framework requires little service recovery time and runtime overhead. Finally, the framework is cost effective since it does not require expensive server replicas. It can restart the service on the original machine with no state lost.

The basic idea of the framework is to achieve the goal of zero-loss service restart, through the cooperation between the service and the operating system. We design the required kernel mechanisms to support a zero-loss service restart. First, we keep the I/O channels that are currently used by the service, and migrate them to the new service instance after the service restart. Keeping the I/O channels is required since the channels are a part of the service state. Second, we hold the unprocessed data that has been read into the memory space of the old service instance in the kernel space. After the service restart, the data can be read and processed again by the new service instance. Third, we design a kernel-level automatic service-restarting procedure, which allows the administrator to keep away from the service recovery job. The procedure is triggered by a fault detection mechanism that can detect the transient and software aging fault.

Our framework needs cooperation from the service side. Specifically, inspired by

the concept of crash only software [5, 6], we require the service store its state a long-live, dedicated state storage provided by the kernel. This enables the service state to be safely migrated to the new service instance. In addition, a service should implement a recovery procedure to store its state according to the content in the state storage. Although modification to the service code is required, the modification is straightforward. Since each service already maintains the service state in its internal data structures, storing/restoring the state to/from the state storage is trivial. This allows our framework to be applied on most of the existing Internet services.

We implement the kernel supports in a Linux kernel module. Moreover, we adopt the service-side cooperation on a popular tiny web server, thttpd [1]. According to the performance results, the runtime overhead of our framework is less than 6.8%. And, the throughput degradation due to the service recovery is only 8.5%. This proves that our framework is efficient for Internet services.

1.2. Thesis Organization

The rest of the thesis is organized as follows. We describe the related work in Chapter 2, which is follow by the detailed illustration of our kernel support mentioned in Chpater3. In Chapter 4, we propose several programming guidelines, and describe how we apply the framework on *thttpd*. Chapter 5 shows the performance result of our framework. Specifically, the overhead and restart effect are measured. Finally, we conclude and discuss future work in Chapter 6.

CHAPTER 2

RELATED WORK

In this chapter, we describe the previous works that were used or can be used for building fault tolerant Internet service systems.

Checkpointing [19] is one of the most well known approaches for system recovery. It checkpoints the software state into a stable storage. When a fault happens, the system can be recovered from the last checkpointed state. This approach can be applied on different levels, such as user library level [20, 24], compiler level [13, 14, 15], operating system level [9, 11], or hardware level [21]. Although this approach can recover a system from transient faults, it can not deal with the software aging problem. This is because the checkpointed state is aged, instead of fresh. Therefore, a fault happened due to the software aging will appear again immediately after the last checkpointed state is restored. Another drawback of this approach is that many checkpoint techniques incur large overheads since the large amount of the checkpointed state and the access to the stable storage.

The concept of developing recovery-oriented software for dealing with errors is proposed by the Recovery-Oriented Computing (ROC) project [17], which is a joint effort of U. C. Berkeley and Stanford University. Different from the previous research that usually addressed on the Mean Time to Failure (MTTF), ROC offered high availability by reducing the Mean Time to Repair (MTTR). In ROC, several techniques that are related to ours were proposed. Rewind-Repair-and-Replay (3R) [4] tries to recover the errors caused by the administrators. As the name indicates, when a fault happens, it rewinds the system to a state before the error occurs, tries to repair

the error, and then replays the operations. Since we address on the transient faults and the software aging problem, there is no need to rewind and repair. Therefore, service restart is more suitable for addressing the problems than the 3R approach.

Recursive Recovery (RR) [5, 6] is another technique proposed by ROC. Similar to our work, it addresses the transient faults and the software aging problem by using the software restart mechanism. Under this approach, a service is made up of fine-grained and loosely-coupled components. And, only the faulty components are restarted when a fault occurs, which speeds up the restarting process. The limitation of this approach is that it requires the system to be made up of fine-grained and loosely-coupled components. Many existing Internet service programs are not able to satisfy the requirement. Moreover, the inter-component communication will degrade the system performance. By contrast, our approach doesn't have the requirement and is more feasible for existing Internet services.

Scalable Network Services (SNS) [7] presents an architecture that supports scalable and fault tolerant Internet services. Similar to RR, a service is made up of several components. And, component failures can be detected by other components. However, SNS assumes that a component only have soft state, which can be reconstructed after the component restart. This is not valid for most of the existing Internet service programs. For example, a web server should have the state of the on-line requests at the user level and the open file information (including the TCP connection information) at the kernel level. Generally, the state is lost if the web server process crashes.

Service Continuation [22] allows an on-line client session to be migrated from one server to another cooperative server. Similar to our work, it uses an application-kernel

cooperation approach for session state migration. However, it relies on a connection migration protocol such as M-TCP [23], which requires modifications to the client-side TCP. Moreover, it copes with network congestion or server overload conditions and doesn't deal with server failures.

FT-TCP proposed a method to resolve the faults on TCP-based Internet services. It inserts wrappers around the TCP layer to log the state of the TCP connections associated with the service. If the system crashes, the TCP connections are reestablished. And, the service state is reproduced by running a new copy of the service application from the beginning and feeding it with the logged I/O requests. That is, it replays the process before the server crashes. The advantage of this approach is that it does not need to modify the service applications. However, the replaying process may take a long time. We eliminate the need for replaying the service application and reestablishing the TCP connections since we address on transient and software aging faults happen on the service itself and we take an application-kernel cooperation approach. This results in much better performance, especially for popular Internet services.

There are still some research efforts that address on the fault tolerant web services [2, 25, 26]. They use the server replication approach to improve the availability of the web server. When the primary server fails, the on-line requests can be migrated to the backup server. These approaches regard the service unavailability of a server as a node failure and thus use the backup node to take over the following jobs. By contrast, we address on the transient faults and the software aging problem happen on the service itself. Therefore, we take a more efficient and cost effective approach that restarts the service on the original node. We keep the state and the I/O channels of the

service, and hand over them to the new service generation to achieve the goal of zero-lost service restart.

CHAPTER 3

KERNEL SUPPORTS OF ZERO-LOSS SERVICE RESTART

As shown in Figure 3.1, the life cycle of a general restartable service consists of four kinds of phases. The first phase is the *service initialization* phase. In this phase, the service initializes their serving environment, and then the service will enter a stable state to wait for providing services. The second phase is *service execution* phase. In this phase, the service serves requests until a fault occurs. The third phase is *process restarting* phase. In this phase, kernel detects the fault of the service and restarts a new instance of the service with the same non-broken I/O channels. And, the forth phase is the *service recovery* phase. In this phase, the restarted service restores its state information. After the service recovery phase, the restarted service goes back to the service execution phase. It continues serving the on-processing requests when the fault occurs, and starts accepting new requests. From the figure we can see that, a process generation begins when the process is started (created or restarted), and ends when the process is terminated (either normally or abnormally). A process generation may consist of the service initialization and service execution phases (i.e., for the first generation), or the service recovery and service execution phases (i.e., for the the other generations).

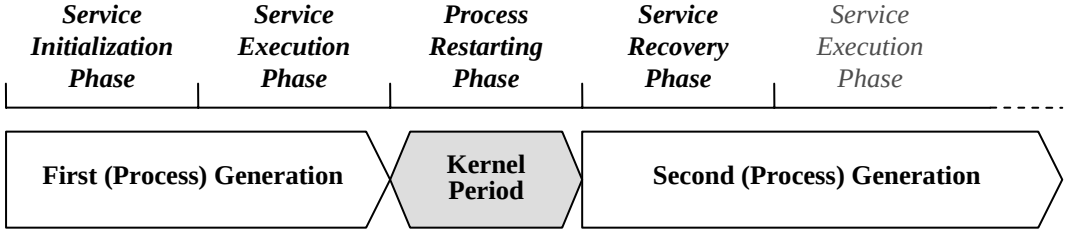


Figure 3.1: Life Cycle of a General Restartable Service

Existing APIs exported by operating systems have some problems to achieve the goal of zero-loss service restart. Firstly, the detection and restarting of a failed process is hard to be fully automatic. It requires the administrator to manually restart a new process. Secondly, no matter a process exits normally or abnormally, its open files will be closed by kernel. This will break all of the channels between service and the outside world. Moreover, even if the open files can be kept, the unprocessed data that was read into the failed process is still hard to be recovered. This is true for non-storage-based files, such as pipe, sockets and etc. When a read is issued on such a file, the data will be copied to the user buffer, and the kernel-level buffer will be freed. If the process fails at this time, the data will be lost. Therefore, to achieve the goal of zero-loss restart, some additional kernel support is needed.

In this paper, we propose three kernel functionalities that can be integrated into an operating system to support zero-loss service restart. The functionalities are *fault detection, I/O channel keeping,* and *restart management*. In addition, we add three system calls that allow user-level service programs to register their restartable service and holding the unprocessed data.

3.1 FAULT DETECTION

When a fault cause a service to crashes, the operation system will terminate the service process and release the resources owned by the service. Therefore, our fault detection mechanism is based on the interception of the abnormal process termination. Specifically, the fault detection is implemented by intercepting the *do_exit()* function to check the *error_code* parameter in linux kernel. 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). Therefore, we can simply check the bit to see if the process termination is abnormal.

However, the software transient fault and software aging are not the only way to go above path. Some kinds of software bugs will go through the same path but will soon occur again after the faulty process is restarted. These kinds of software bugs are not suitable to resolve by restarting the faulty process. Therefore, in order to avoid intercepting these kinds of software bugs we add a per service process variable, *last_restart_time*, to record the last time that the service process was restarted. During fault detection, we consider it is a transient fault if the difference between current time and *last_restart_time* of the service process is large than a certain threshold (currently, 1 minute). Oppositely, we consider it is a software bug if it is too soon to need to be restarted.

In our system, a service has to register its information to the kernel if wants to be zero-loss restaratable. When the fault occurs in a restartable process, the kernel uses the registration information to create a new generation of the process and performs the recovering job. The information can be registered through a *reregist()* system call, which is shown in Figure 3.2.

```
ssize_t reregist ( char* bin_path,
                   char* argv[],
                   char* envp[]
                   int child_id,)
```

Figure 3.2: Prototype of Restart Registration System Call

The first three parameters represent the path and arguments of the service program. And, the *child_id* parameter is used to identify the role of the current

process. In a multi-process service, different processes may play different roles and hence require different action for recovery. By registering the role of the current process, the restarted process can perform the recovery actions accordingly. In general, the child_id can be set as 0 for a single-process service. For a multi-process service, the developer can specify a unique child_id for each child process.

In addition, we also provide a *getreginfo()* system call so that the service process can retrieve the its registration information. This is primary used by the next service generation to get the registration information of its previous generation. The detail usage of these two system calls will be described in Section 4.1.2 and Section 4.1.3.

3.2 KEEPING I/O INFORMATION

To achieve the goal of zero-loss restart, the I/O channels of a service should not be terminated when a service crashes. Instead, they should be kept and handed over to the next service generation. Keeping I/O channels consists of two tasks. One is to prevent the I/O channels from being closed, and the other is to keep the input data in these channels. We discuss these two tasks in Section 3.2.1 and Section 3.2.2, respectively.

3.2.1 I/O Channel Keeping

The I/O channels are regarded as files in most Unix-like operating systems, including Linux. For each process, the Linux kernel maintains an open file table for it, which contains all the opened files of this process. Therefore, we can keep the I/O channels of a service process by preventing all the files in the corresponding open file table from being closed. A simple approach for keeping the I/O channels is to increase

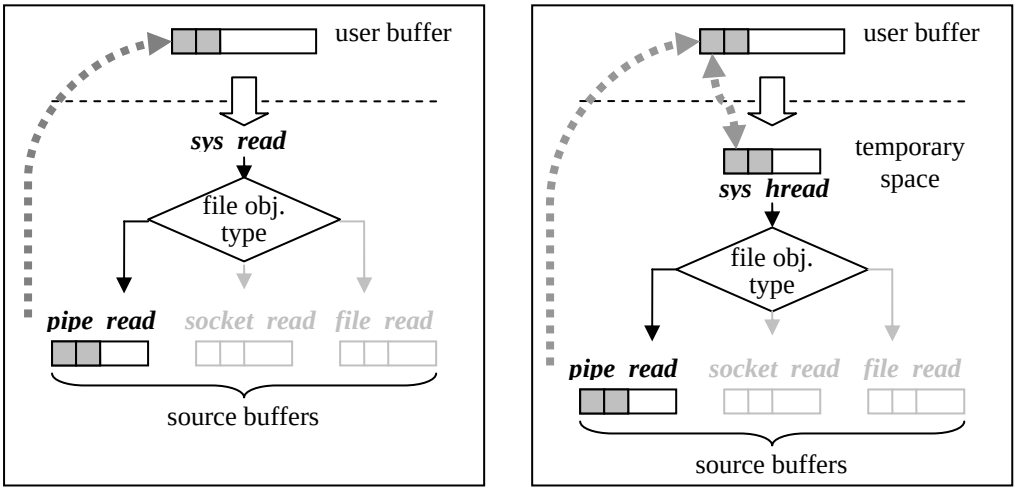
the reference count of the open file table. As a result, the kernel will not close the files and frees table when the faulty process exits. When the service is restarted we can hand the table to the new service generation so that it can continue providing service with those I/O channels.

3.2.2 Data-Holding Read Operation

As we mentioned above, a process failure will cause, all the data that has been read into the process memory space but not yet been processed to be lost. To avoid this problem, we propose an alternative read operation, namely data-holding read. Figure 3.3(a) shows the comparison between original read operation and data-holding read operation. From the figure we can see that, the original read operation copies data from the per-file source buffer to the user-specified buffer. For a non-storage-based file object such as pipe or socket, the copied data in the source buffer will be freed. As a result, the data will be lost if the user-level service crashes at that time. Therefore, the original read operation is not sufficient for supporting zero-loss service restart. Specifically, if a process fails with some unprocessed data left in it, the data will be lost.

Figure 3.3(b) shows the data-holding read operation. The difference between the read and the data-holding read operations is that data is removed explicitly in the latter. That is, the user program can specify not only how many bytes to read but also how many bytes to delete. To achieve this goal, we maintain a kernel level temporary space between the per-file source buffer and the destination user buffer. This temporary space is used to keep the data which has already been read to the user buffer but has not been specified to be deleted. Thus, already-read data can be

retained in the temporary space until the user program doesn't need it.



(a) Original Read (b) Data Holding Read

Figure 3.3: Comparison between the original read and data-holding read operations

Figure 3.4 shows the prototype of data-holding read system call. Users invoke it to read *rlen* bytes of data from the file *fd* to the user buffer *buf*, as well as to delete *dlen* bytes of data from the kernel temporary space.

```
ssize_t hread( unsigned int fd,
               char * buf,
               size_t rlen,
               size_t dlen )
```

Figure 3.4 Prototype of Data-Holding Read System Call

The kernel-level temporary space is implemented as the per-file *dhr* (*data-holding read*) structure. A *dhr* structure is created for a file when the user program invokes *hread()* on that file at the first time. Figure 3.5 lists the fields in the *dhr* structure and the related data structures.

```
struct dhr {
        struct file* file;
        unsigned int read_pt;
        unsigned int destroy_pt;
        unsigned int write_pt;
        struct dhr_buf* dhr_buf_head;
};
```

```
# define DHR_ONE_BUFFER_SIZE 4096
struct dhr_buf {
        struct dhr_buf* prev;
        char buf[DHR_ONE_BUFFER_SIZE];
        struct dhr_buf* next;
};
```

Figure 3.5: Definition of the *dhr* structure

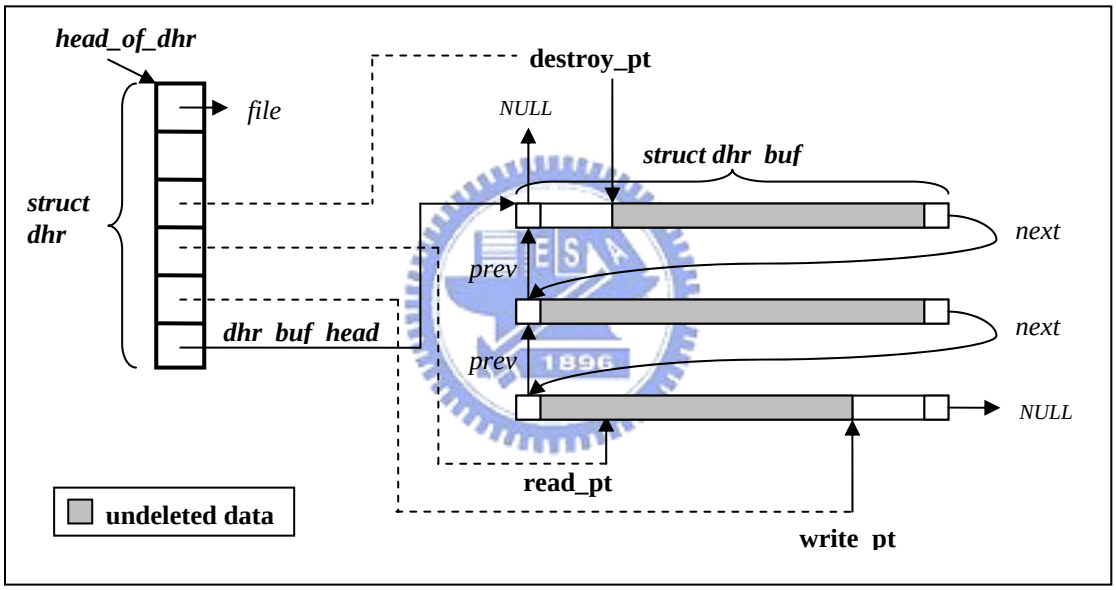


Figure 3.6: Structure *dhr* and structure *dhr_buf*

Each *dhr* structure has a pointer to its corresponding file, and a pointer to a chain of fix sized buffers (i.e., *dhr_buf*) for holding the already-read data. In addition, the *dhr* structure uses three pointers to manage the buffer chain, as shown in Figure 3.6. The *read_pt* and *destroy_pt* point to the last bytes of data that has been read and deleted by the user process, respectively. Therefore, the data between *destroy_pt* and *read_pt* represents the bytes that are already been read but not been deleted by the user program. The *write_pt* indicates the number of bytes have been copied into the

dhr. It is worth to note that the values of *read_pt* and *write_pt* may be different. We will discuss it later in this section.

At the end of this section, we describe the procedure of the data-holding read system call, which is shown in Figure 3.7.

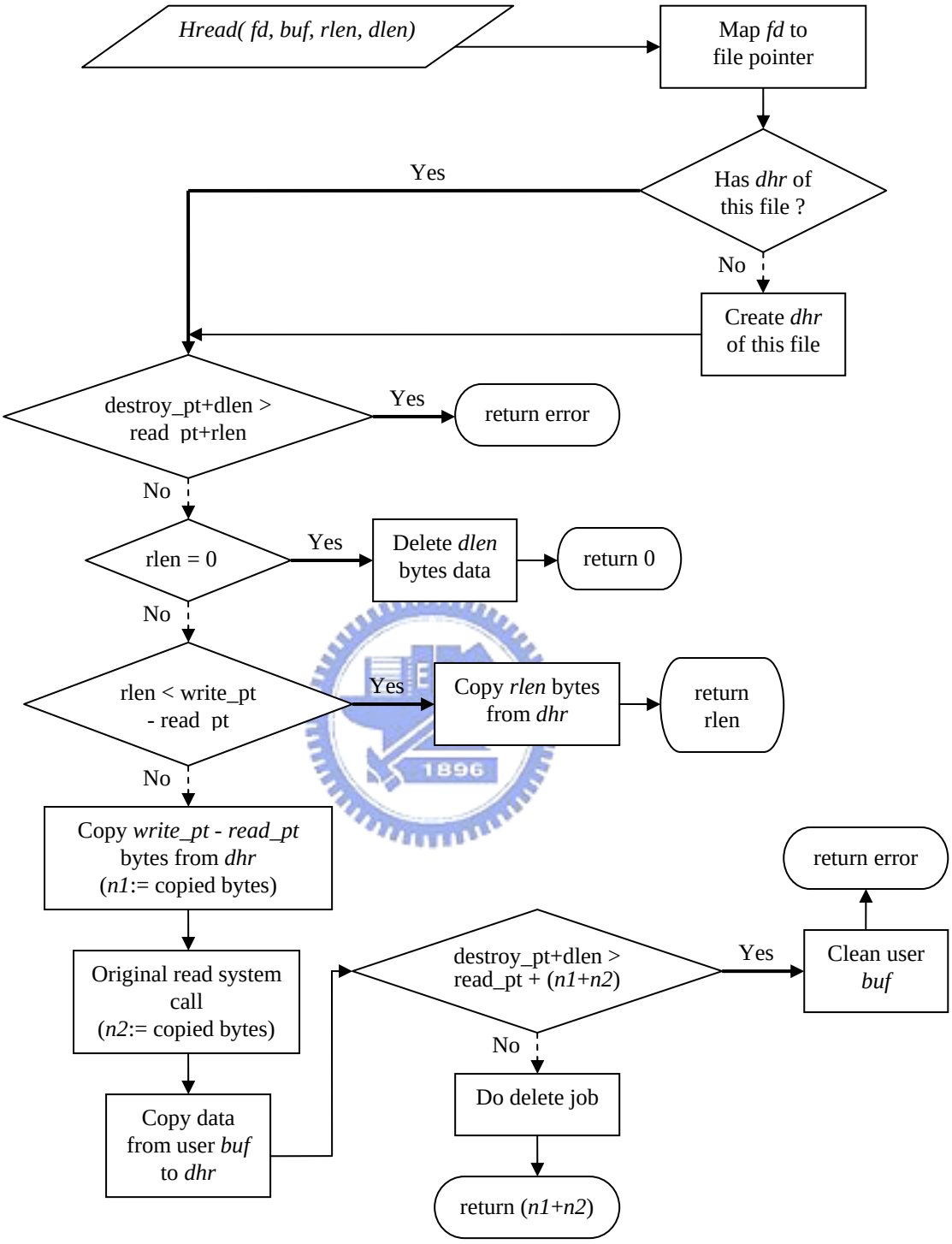


Figure 3.7: Flowchart of Hread() Function

When a user program calls the system call, the kernel locates or creates the *dhr* structure for the file, and then performs the following procedure.

1. The kernel checks to see if the user program wants to delete more data than it can

read (i.e., dlen+destroy_pt > rlen+read_pt). If it does, the kernel just returns an error.

2. If *rlen* is 0, the kernel just deletes *dlen* of data in the temporary space and returns 0.
3. The kernel checks if there is enough unread data in the *dhr* buffer chain (i.e., rlen < write_pt - read_pt).
   3a. If there is, the kernel tries to copy *rlen* bytes of data from temporary space to the user buffer. And then, the kernel updates the *read_pt*, and returns *rlen*.
   3b. Otherwise, the kernel copies all the bytes between *write_pt* and *read_pt* and performs the original read function to read the remaining data from the per-file source buffer to the user buffer. Since the original read operation copies data directly to the user buffer, we have to copy the data from the user buffer into the *dhr* buffer chain in order to hold the data in the temporary space.
4. The *write_pt* is updated accordingly.
5. The kernel checks to see if the data deletion goes beyond the available data (i.e., dlen+destroy_pt > write_pt).
   5a. If it does, the kernel clears the user buffer and returns an error. This buffer clearing is necessary since we want to preserve the all-or-none semantic of the system call. Note that, in this case, the *read_pt* and *write_pt* will become different.
   5b. Otherwise, the kernel updates the *read_pt* to the value of the *write_pt*, deletes *dlen* bytes of data, and update *destroy_pt* accordingly. Once the *destroy_pt*

goes beyond a buffer in the buffer chain, the buffer will be freed.

6. At last, the kernel returns how many bytes that have been copied to the user program.

During the process restarting phase, the kernel will set the *read_pt* as the *destroy_pt* for each file opened by the user service. This allows the restarted service it to read the data that has not been completely by its previous generation.

3.3 RESTART FLOW

The fully automatic and zero-data-loss restart is controlled by the restart manager. Figure 3.8 shows the restart flow.

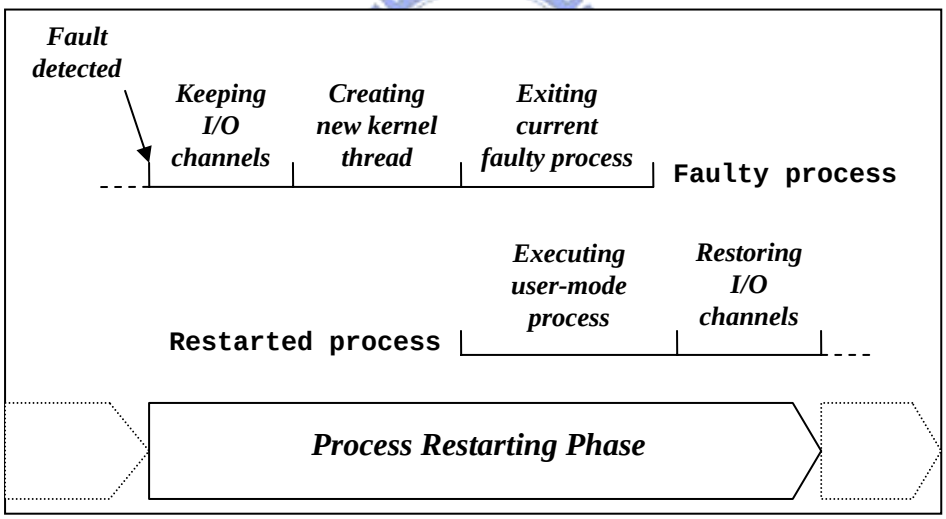


Figure 3.8: Restart Flow

First of all, a fault of the service process is detected by the fault detection routine. Then, the kernel performs the I/O channel keeping operation to prevent the I/O channels of the service process from being closed. For all the I/O channels, the kernel rewinds the *read_pt* in the kernel temporary space to the value of the *destroy_pt*.

A kernel thread is then created, which will eventually become the next generation of the failed service process. At the moment, the faulty process can be

terminated. And, the kernel thread invokes the *exec_usermode_helper()* function to turn itself into a user-level process and execute the binary image of the service. As a result, a new generation of the service is started. Finally, the I/O channels can be handed over to the new service generation. This is done by copying the kept pointer that reference to the open file table into the task control block of the new service generation.

3.4 EXPERIMENTAL RESULTS OF THE KERNEL SUPPORT

In this section, we present the experimental results of the kernel support. Since buffers in the kernel temporary space are allocated/deallocated on demand, small buffer size will increase the number of allocation/deallocation and degrades the performance of *hread()*. In this experiment, we measure the impact of the buffer size (i.e., DHR_BUFFER_SIZE) on the performance of *hread()*. We use a small test program that reads a file through *hread()* with different buffer sizes, and record the resulting times in CPU ticks. Figure 3.9 shows the result. As shown in the figure, the performance improves as the buffer size increases. However, the improvement becomes little when the buffer is larger than 4096 bytes. As a result, we choose 4096 as the buffer size in current implementation.

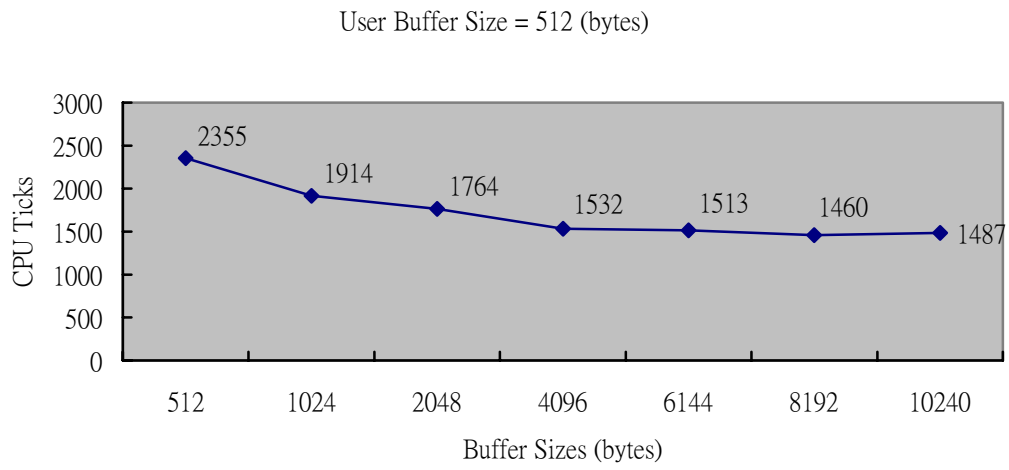


Figure 3.9: Performance of Hread() under Different DHR_ONE_BUFFER_SIZE

Figure 3.10 compares the performance of the original *read()* and the *hread()* system calls under different user buffer sizes that the test program specifies. From the figure we can see that when the user buffer size is smaller than 512 bytes, the overhead of *hread()* ranges from 36% to 46%. And, with the increase of the user buffer size, the overhead grows. The largest part of the overhead happens in the *copy_from_user()* function, which is a standard function used by the linux kernel to copy data from a user-mode buffer. And, we use the function to copy the read data from the user buffer to the kernel temporary space. According to our measurement, the execution time of the function grows rapidly as the data size increases. Although the overhead seems to be large, we still consider it be acceptable because that the reading frequency of non-storage based files is usually far less than the writing frequency for many Internet services, such as web services, FTP service, streaming service, and etc. We will verify it in Section 5.2.1.

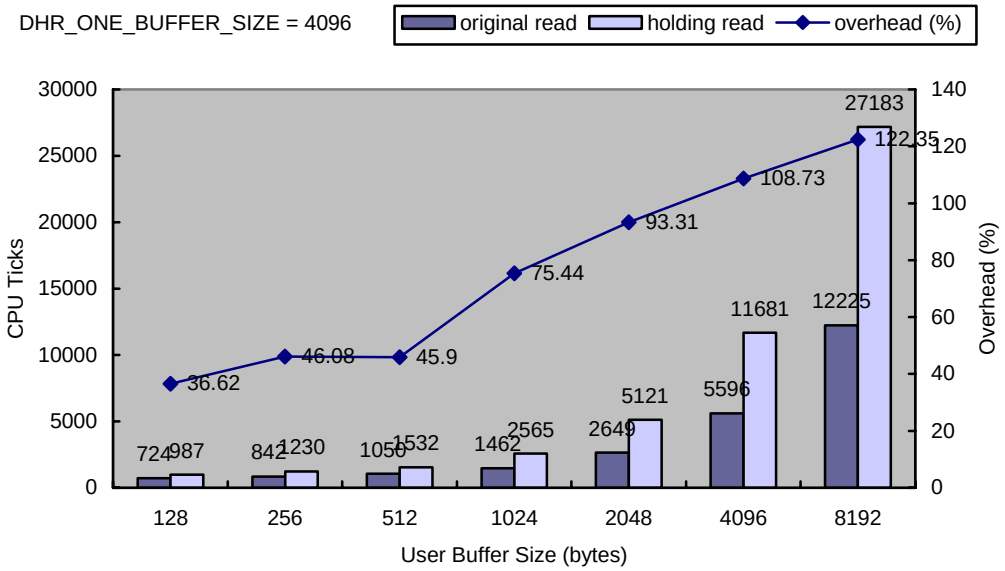


Figure 3.10: Performance Comparison of Read() and Hread() under Different User Buffer Sizes

Besides hread(), we also measure the execution time of the other two system calls we added (i.e., *reregist()* and *getreginfo()*). Figure 3.11 shows the result. Both system calls require less than 2 us.

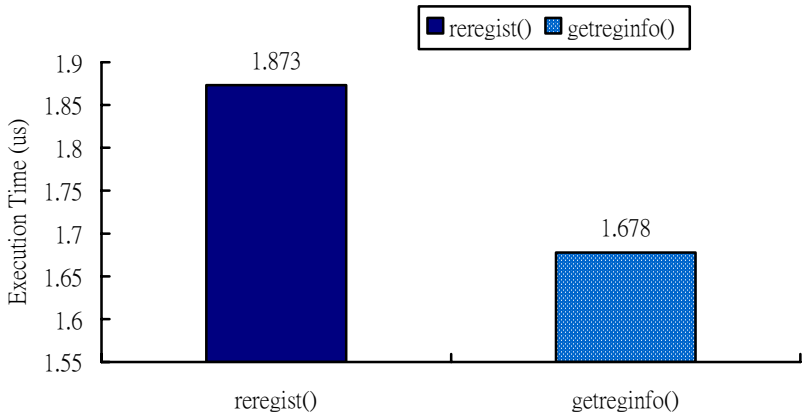


Figure 3.11: Time Complexity of Reregist() and Ggetreginfo() System Call

At last, a breakdown of the kernel execution time spent for starting a new service generation is given in Table 3.1. The time begins when the fault is detected and ends

at the end when the I/O channels are restored. From the table we can see that, keeping and restoring I/O channels is quite efficient (specifically, only about 7.4 us). Most of time is spent on creating the kernel thread for resuming the new service generation and executing the *exec_usermod_helper()* function.

	I/O channel keeping	Kernel thread creating	User-mode process executing	I/O channel restoring
Execution Time	*7.294 us*	*15.264 us*	*185.78 us*	*0.125 us*

Table 3.1: Kernel-level Execution Time for Restarting a New Service Generation

CHAPTER 4

PROGRAMMING GUIDELINES OF ZERO-LOSS RESTARTABLE SERVICE

The kernel support that we described in Chapter 3 provides a basics building block for a restarable environment. However, cooperation from the service developers is also needed to achieve the goal of zero-loss service restart. The cooperation includes the following jobs. First, the service developers have to use the system calls mentioned in Chapter 3 to register/retrieve the service information, and hold the useful data in the kernel. Second, the developers should use a dedicated state storage to store the service state. Third, the service developers should follow the programming guidelines mentioned in Section 4.2 to facilitate the recovery procedure at *service recovery* phase.

In Section 4.1, we will illustrate how to use those kernel supported system calls mentioned in the previous chapter. And, we will also propose a model that allows two successive service generations communicate through a dedicated storage. In Section 4.2, we describe the programming guidelines.

4.1 PROGRAMMING STYLE OF RESTARTABLE SERVICE

4.1.1 Using *Hread()* System Call

```
void handle_new_connection() {
     int read_fd;
     char* buf = malloc(…);
     char* request = malloc(…);

     read_fd = socket(PF_INET, SOCK_STREAM, 0);
     bind(read_fd, …);
     listen(read_fd, …);
     ...

     int sz, count = 0;
     while( !read_full_request(request) ) {
          if( (sz = hread(read_fd, buf, 10, 0)) > 0) {
               strncpy((char*)(request+count), buf, 10);
               count += sz;
          }
     }
     process_rcv_data(request);
     hread(read_fd, buf, 0, count);
     ...
     close(read_fd);
}
```

Read a full request

Really process the received data

After processed, delete what you have processed in the kernel temporary space

Close the socket

Figure 4.1: Example pseudo code of using data-holding read system call

In this section we describe how to use the *hread()* system call to avoid data loss when a fault crashes the services. Figure 4.1 shows the pseudo code of handling a request using the *hread()* system call. In the while loop, the program reads a full request from socket *read_fd*. Since the data has not been processed, the forth parameter (i.e., *dlen*) is set to 0 in order to keep the data in the kernel temporary space. After receiving the request, the *process_rcv_data()* is invoked to process the request data and generate the result. Finally, the *hread()* system call is invoked again with the *rlen* parameter set to 0 and the *dlen* parameter set to the data length of the request data. This invocation is used to delete the request from the kernel temporary space.

If a fault occurs before the end of the *process_rev_data()* function, the request

data still remains in the kernel temporary space. As we described in Section 3.3, the kernel will rewind the read pointer to the position of the destroy pointer when the service is restarted. Therefore, after the new generation starts, it can read the request from the kernel temp space and process it. On the other hand, if the fault occurs between the *process_rev_data()* and the second *hread()* function, the new generation has the ability to know that the result was generated. As a consequence, it will delete the request in the kernel temporary space. To know that the result was generated, there must be a communication channel between the successive generations, which is described in Section 4.1.4.

4.1.2 Using *Reregist()* System Call

```
#define CHILD_ID_INIT_VALUE 1

void main(char* argv[]) {
     int cid;

     reregist("/usr/local/sbin/service", 0, argv, NULL);
     ...
     cid = fork();
     if(cid == 0) { // child process starts
          reregist("", CHILD_ID_INIT_VALUE, argv, NULL);
          ...
     }
     else { // parent process
          ...
     }
     ...
}
```

Register the main process of the service

Register the child process as child_id = 1

Figure 4.2: Example pseudo code of using restart registration system call

Reregist() system call is used to register a restartable service. In most of the cases, the developer invokes *reregist()* during the program initialization. Figure 4.2 shows the typical usage of *reregist()* system call. In the figure, the first *reregist()* invocation means that the developer wants the main process of the service to be restartable. And, the second *reregist()* invocation registers the child process as a restartable process. Note that the *child_id* parameter is different in these two invocations. This allows the restarted process to know which child process (or main process) it is.

4.1.3 Use *Getreginfo()* System Call

```
#include <registration.h>
int is_restarted = 0;
void main(char* argv[]) {
     struct reg_info* reg_info = (struct reg_info*)malloc(sizeof(struct
                                                          reg_info));
     if( getreginfo(reg_info) ) {
          is_restarted = reg_info->is_restarted;
     }
     if( is_restarted ) { // recovery path
          switch( reg_info->child_id ) {
               case 0: break;
               case 1: goto child_process_1;
               ...
          }
          ...
     }
     else { // normal execution path
     ...
     }
}
```

Figure 4.3: Example pseudo code of using get registration information system call

Figure 4.3 shows an example of using the *getreginfo()* system call. The most important function of this system call is to tell whether the current process is original or restarted. If it is restarted, the process should execute the recovery path. Otherwise, it executes the normal path. Note that both paths should be programmed by the service developer. And, this system call should be called at the beginning of the program to determine the execution path of this generation. In addition, this system call also returns the *child_id*. With that information, the current process can know the child identifier of its previous generation. Therefore, the developer can divide the recovery path into a number of recovery procedures. Each procedure takes responsible for recovering a child process or the main process. As a result, the new generation can execute the corresponding recovery procedure to recover its previous generation.

4.1.4 Using Shared Memory for State Handover

To achieve the goal of zero-loss service restart, the developers should separate state from logic when developing the service. The service state has to be stored in a dedicated storage, which should be live across successive service generations. This allows the state to be handed over to the new generation. In our system, the dedicated storage is implemented by shared memory.

We use shared memory because of the following reasons. First, the shared memory attached on a process is available until the process detaches it. Therefore, if a service does not detach the shared memory before it terminates, the next generation will be able to attach the same shared memory area. Second, shared memory is efficient so that there will be little performance impact for storing service state in shared memory.

However, there is a problem when using shared memory as the state store. That is, the shared memory had better be attached at the same address for two successive generations. This is because that the service state stored in the shared memory may contain pointers which point to the data in the shared memory. If the shared memory area can not be attached at the same address, the new generation must adjust the pointers in the shared memory. For example, if we store a linked list in the shared memory, the values of all the links should be adjusted when the new generation attaches the shared memory to a different address.

Therefore, the developer should reduce the usage of pointers for maintaining the service state. If there are still some necessary pointers, the developers should write a procedure to adjust these pointers. In order to accomplish the adjustment, the application can store the attached address in a fixed location of the shared memory when it attaches the shared memory. Therefore, the new generation can calculate the difference value of the attached address and update all the pointers in the shared memory accordingly.

4.2 PROGRAMMING GUIDELINES FOR ZERO-LOSS RESTART

In this section, we propose several programming guidelines that make the service operate at its logical level and hence become zero-loss between generations.

Avoid registering the signal handlers of the signals that cause the abnormal termination of the process. Such as SIGSEGV, SIGTRAP, SIGABRT, and etc are this kind of signals. It is in order to let the abnormal termination of the faulty service can be caught by our fault detector instead of the programmer specified function.

Abstract the state variable of the service. The *state variables* contain all the

necessary information that represents the service state during the service execution. A piece of state information should be included into the state variables if the restarted service can't reconstruct the whole service state without it. The state variable of the service needs to be stored in the dedicate storage and be updated when necessary. This allows the service to be executed as a stateless client of the state storage.

Design recovery procedures for service recovery. The recovery procedures will be executed during the *service recovery* phase. In a recovery procedure, the service must reconstruct its state from the state variables. And then, it tries to finish the ongoing jobs of the previous generation when the fault occurs.

Divide the execution into several stages. This can reduce the recovery time. When a stage is finished, the service can record the progress and starts the next stage. When a service restarts, the next generation can go through the unfinished stage as in its normal path. The recovery time is reduced since the jobs in the already finished stages are not needed to be performed again. For example, the page request processing in a web server can be divided into four stages: request reading, request parsing, response header sending, and response body sending. For large responses, the last stage can further be divided into more sub-stages. When the service restarts, the new generation can get the processing progress of the request. If, for example, the first two stages are finished, the new generation can start sending the response header.

CHAPTER 5

CASE STUDY: ZERO-LOSS RESTARTABLE THTTPD WEB SERVER

In this chapter, we present a case study that applies all the operations and program guidelines mentioned in Chapter 3 and 4 to a well-known tiny web server, *thttpd-2.25b* [1], in order to make it zero-loss restartable. We chose web server as the target because of its popularity on Internet. Thttpd is an open source web server, with simple and well-organized code.

In the following, we will briefly describe the design and execution flow of the original thttpd in Section5.1.1. In Section 5.1.2 we will present how we modify the original thttpd to achieve the goal of zero-loss restart. Furthermore, we will analyze the experimental results in Section 5.2.

5.1 ZERO-LOSS RESTARTABLE THTTPD

5.1.1 Original Thttpd

In this section, we will describe the execution flow of thttpd. Thttpd uses single-process implementation of HTTP 1.1 protocol [8], and it divides the procedure of handling a request into two stages, namely reading and sending. Figure 5.1 shows the stage flow of request handling in thttpd.

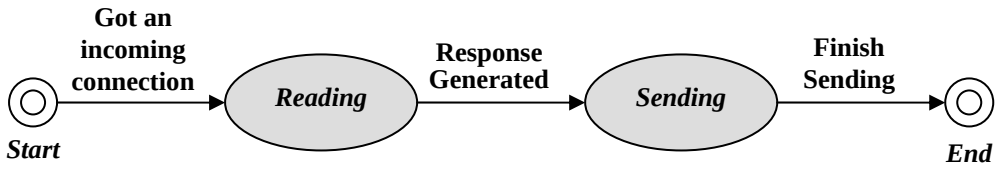


Figure 5.1: Stage Flow of Request Handling in *Thttpd*

When thttpd starts, it creates, binds, and listens on a TCP socket. Then, it probes for incoming requests by performing *select()* on the TCP socket. If there is no request, the server keeps on probing. When the server gets a request, it creates a connection entity structure for this request and the connection enters into the *reading* stage. The connection entity structure contains almost all the information needed to construct the total state variables of thttpd. This will be discussed in the next section. In the *reading* stage, the server reads the request from the client, parses the request, and generates the corresponding response. After the response is generated, the connection enters into the sending stage. In this stage, the server writes out the response to the client.

Note that connections are processed concurrently. Different connections may be in different stages. Moreover, request probing and processing are also handled concurrently. It is worth to mention that, *reading* and *sending* stages in thttpd are divided into more fine-grained sub-stages. In *reading* stage, the server doesn't perform blocking read operation. Therefore, several read operations may be needed to get the full request. And the server will try to handle other requests between two successive read operations. Similarly, in the *sending* state, the server will also try to write a part of the response and then handle other requests or accept new requests.

5.1.2 Zero-Loss Restartable Thttpd

In this section, we describe how we modified the thttpd to make it zero-loss restarable. The modified version is called ZLR_thttpd. First of all, we replaced all socket read operations in thttpd with *hread()*, and applied the *reregist()* and the *getreginfo()* on thttpd. We will not describe the detail of these modifications because that they are simply like what we have mentioned in Chpater 4.

The other modifications are described in the following. First, we identified the state variables of thttpd. In the previous section, we mentioned that the connection entity structure is used to represent a connection in thttpd, and it contains all the information of an on-processing connection. Therefore, we can get the state variables of a connection by extracting the fields in this structure that are necessary for the recovery procedures. Instead of separating the original data structures in thttpd into state variable part and no_state variable part, we make a copy of all state variables and store the copy into the shared memory. And, we update the variable in shared memory before the corresponding variable in thttpd is modified. These can avoid large modifications to the original thttpd.

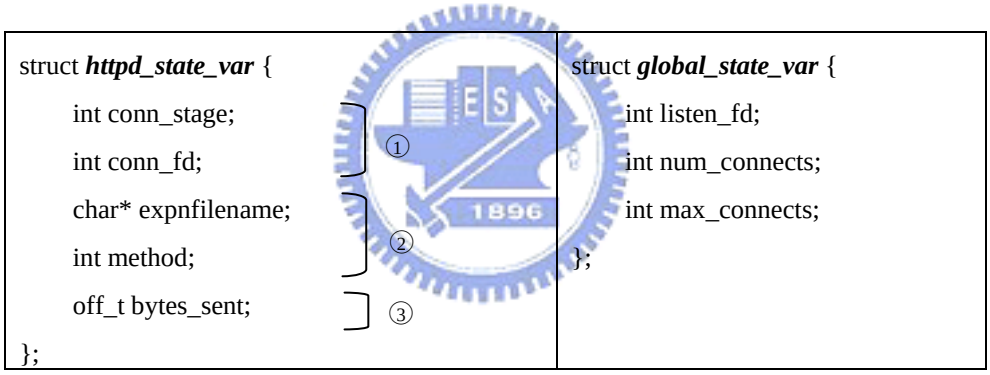


Figure 5.2: State Variables of Thttpd

Figure 5.2(a) shows the per-connection state variables, the http_state_var structure. The fields in this structure can be divided into three parts. The first part contains the most important fields of a connection, *conn_stage* and *conn_fd*. The *conn_stage* field represents the connection stage, and the *conn_fd* field represents the data socket used for communicating with the client. The second part is updated during the *reading* stage of a connection. The *expnfilename* represents the requested file name in the server, and the *method* field indicates the HTTP method. These two fields are generated after the request is parsed. The last part is updated during the *sending* stage. The *bytes_sent* field represents how many bytes of response have been written

to the data socket. In addition to the per-connection state variables, the global variables are maintained in the global_state_var structure as shown in Figure 5.2(b). The *listen_fd* represents the socket that the server uses to receive requests from the clients. The *num_connects* field indicates how many connections are currently processed in the server. And, the *max_connect* field indicates the maximum number of connections that the server can process simultaneously.

The second modification we made was using shared memory areas to store the *state variables* of thttpd. Totally, four shared memory areas are used. As Figure 5.3 shows, these four areas are pointed by four pointers, *shm_pointers*, *httpd_state_vars*, *char_area*, and *global_vars*, respectively.

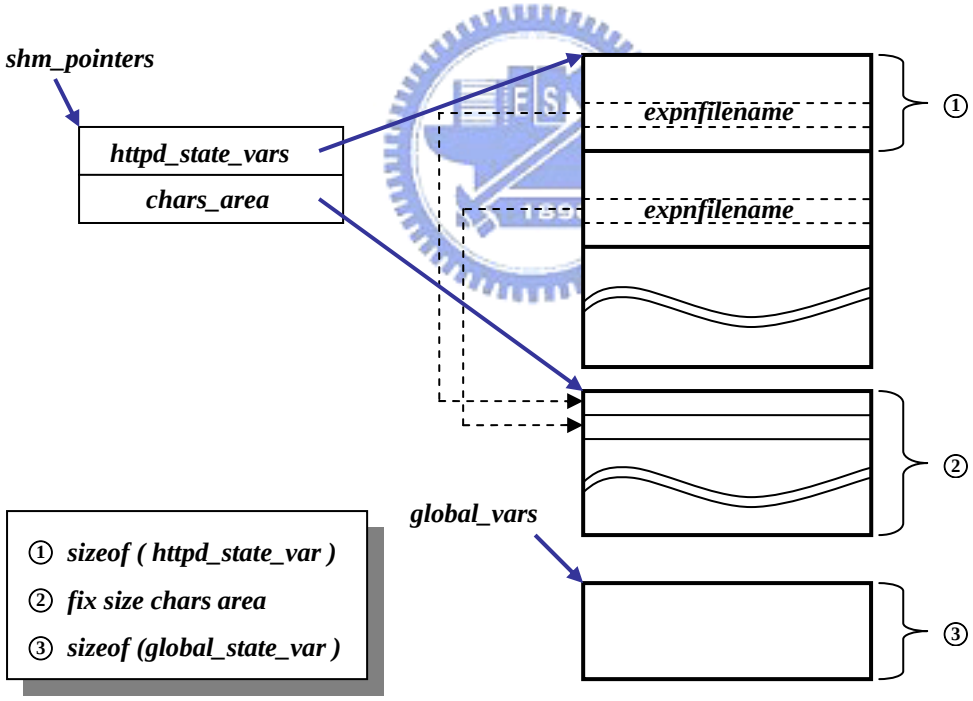


Figure 5.3: Four Shared Memory Areas in Modified *Thttpd*

The area pointed by *httpd_state_vars* is used to store the *httpd_state_var* structures of all the connections. The *char_area* points to a fix-sized shared memory area that is used to store the *expnfilename* fields of all the *httpd_state_var* structures. The above two pointers are stored in another shared memory area, which is pointed by

the *shm_pointers* pointer. In addition, the *global_vars* points to a shared memory area that stores the *global_state_var* structure of the server.

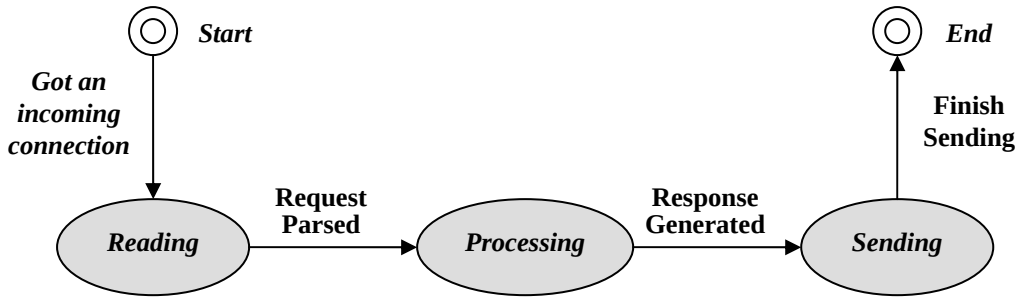


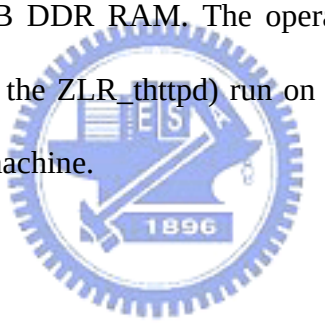
Figure 5.4: State Diagram of Connection in ZLR *thttpd*

The final modification we made was to add a recovery path which will be executed during the *service recovery* phase. Since thttpd is a single-process application, the path contains a single recovery procedure only. In order to facilitate the recovery, we added a new stage, *processing*, between the *reading* and *sending* stages. As shown in Figure 5.4, the stage is entered when a request is completely read and parsed. In this stage, the server will use the parsing result of the request to generate the response. This stage is added to prevent the service going back to the reading stage while the request has been parsed. The recovery procedure performs two jobs. First, it restores all the variables from the shared memory area. Second, it handles the recovery of each on-processing connection according to the connection stage. For the *reading* stage, it tries to finish the reading and parsing job. For the *processing* stage, it tries to read the requested file again into the memory space of the thttpd. For the *sending* stage, the recovery procedure reads the requested file again into the memory space of the thttpd, and then sends the response to client. Note that only the remaining response will be sent since we have recorded the number of sent bytes.

5.2 EXPERIMENTAL RESULTS

In this section, we evaluate the performance of ZLR_thttpd. We first compare its performance with that of the original thttpd when no fault occurs. Then, we verify the functionality of our restart mechanisms by injecting a fault, and we measure the performance of ZLR_thttpd with that fault occurs. The performance is measured by using the WebStone [16] benchmark version 2.5 with the standard testbed profile.

Our experimental environment consists of a client and a server machine, which are connected through a 1Gigabit Ethernet link. Each machine has an Intel 1.6GHz Pentium 4 CPU with 256 MB DDR RAM. The operating system is Linux kernel, version 2.4.18. The thttpd (or the ZLR_thttpd) run on the server machine, while the WebStone runs on the client machine.



5.2.1 Overhead

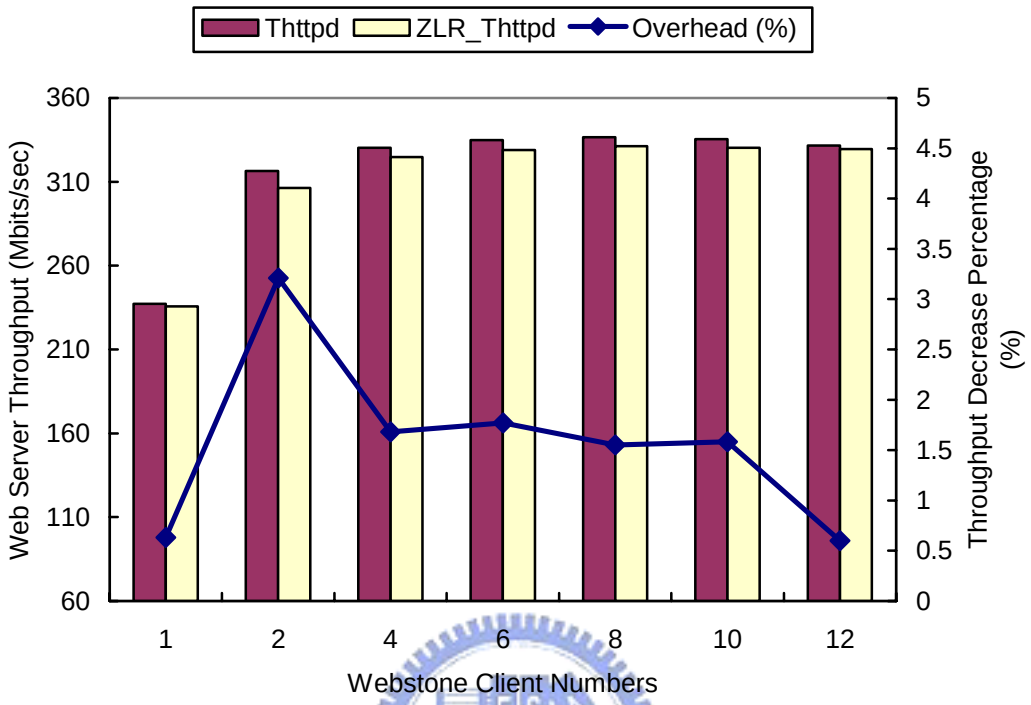


Figure 5.5: Throughput Comparison between the Original Thttpd and ZLR_Thttpd.

Figure 5.5 shows the throughput comparison between the original thttpd and the ZLR_thttpd. The x-axis represents the number of WebStone clients. Each client will establish a large number of connections with the server during the experiment time. The y-axis indicates the server throughput numbers reported by Webstone. From the figure we can see that, our framework results a little throughput degradation. The overhead comes from the backup of the state variables to the shared memory and the using of the *hread()* for socket reading. Besides, the figure also shows the overhead of the ZLR_thttpd, so that we can see the overhead ranges from 0.6% to 3.2%. We consider that it is acceptable.

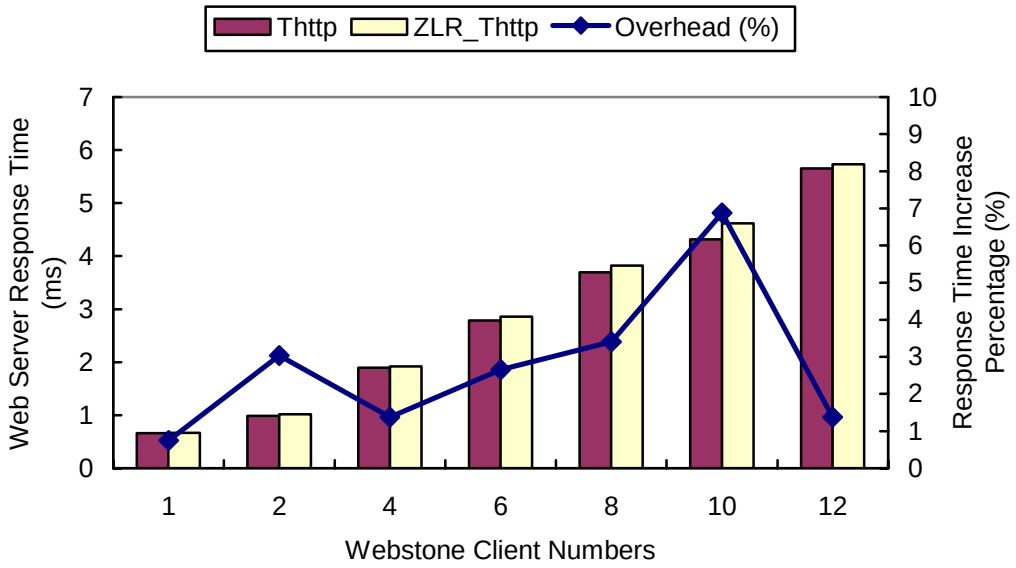


Figure 5.6: Response Time Comparison between the Original thttpd and ZLR_Thttpd.

Figure 5.6 compares the response time between the original thttpd and the ZLR_thttpd. From the figure we can see that, there is little difference between the two. The figure also shows the response time differences, which range from 0.75% to 6.88%. We consider that it is also acceptable.

According to the results of the above experiments, we show that our framework has low runtime overhead when it is applied on a web server.

Webstone Client Number	10	20	30	40	50	60	70	80	90	100
Space Overhead(bytes)	*30160*	*36920*	*39000*	*49400*	*49400*	*59800*	*60320*	*66560*	*78000*	*92040*

Table 5.1: Space Overhead Caused by the Kernel Temporary Space

In addition to the timing overhead, we also measure the space overhead caused by the in-kernel temporary space. Table 5.1 shows the required space under different client numbers. Although the space overhead increases as the client number grows, it still remains small for large client numbers.

5.2.2 Restart

In this section, we first verify the functionality of our framework. Specifically, we show that ZLR_thttpd can be restarted with no state lost when a fault occurs.

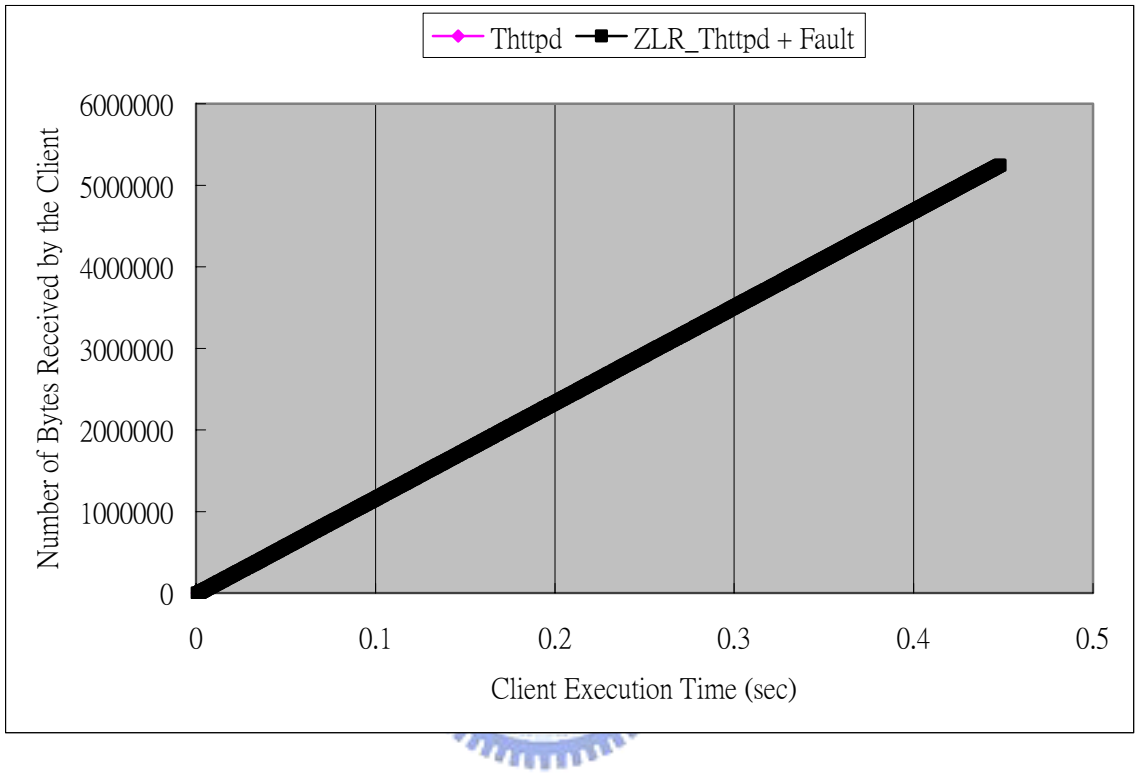


Figure 5.7: The effect at client side when a server fault occurs

In this experiment, we show that the ZLR_thttpd can keep on serving an online client once a fault occurs. We make the client issue a connection and request a 5Mbyte web page from the server. During the experiment time, we inject a fault by sending a segment violation signal to the server. This triggers the restart mechanism. Figure 5.7 shows the result.

The fault is injected after the server has sent 3000000 bytes of the response. As shown in the figure, the server can continue serving (with a new generation) after a fault occurs in it. Remarkably, there is nearly no slow down on the data receiving when a fault happens. The reason is that the TCP connection is still alive when the

ZLR_thttpd fails. With the fast restart of the ZLR_thttpd and the TCP layer buffering, the server-side TCP can keep sending data to the client before the ZLR_thttpd restarts.

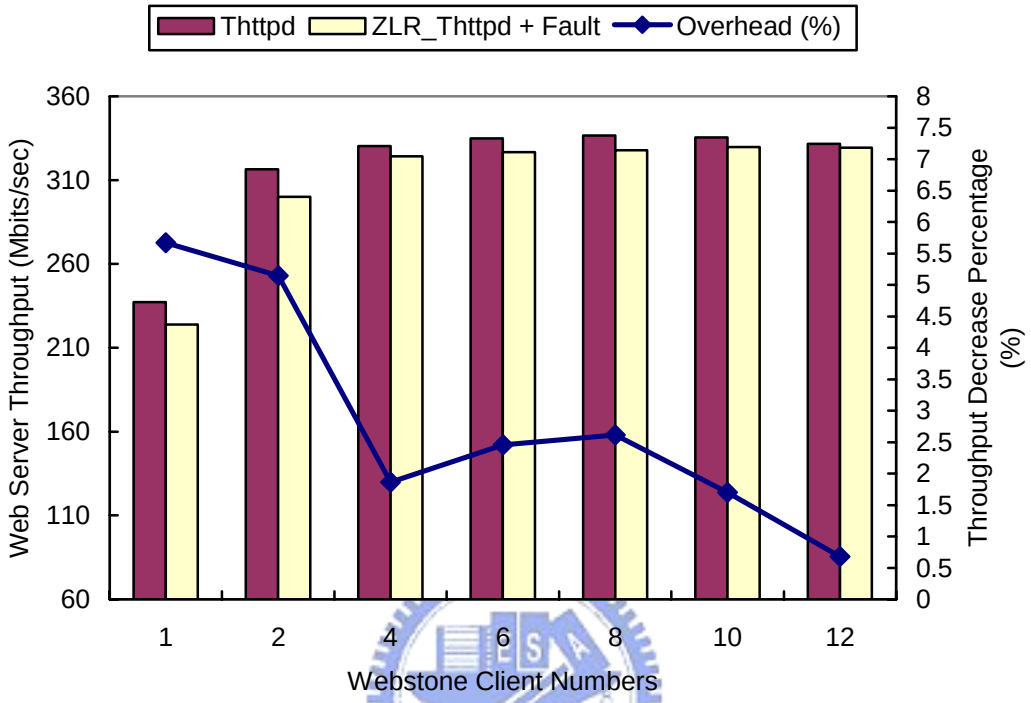


Figure 5.8: Throughput Comparison between Original Thttpd and ZLR_Thttpd with fault.

Figure 5.8 shows the throughput comparison between the thttpd and the ZLR_thttpd when a fault occurs. From the figure we can see that, ZLR_thttpd only result in little throughput degradation. The difference ranges from 0.68% to 5.67%, which is similar to that reported in Figure 5.5. As we mentioned before, this is due to the fast recovery and TCP buffering.

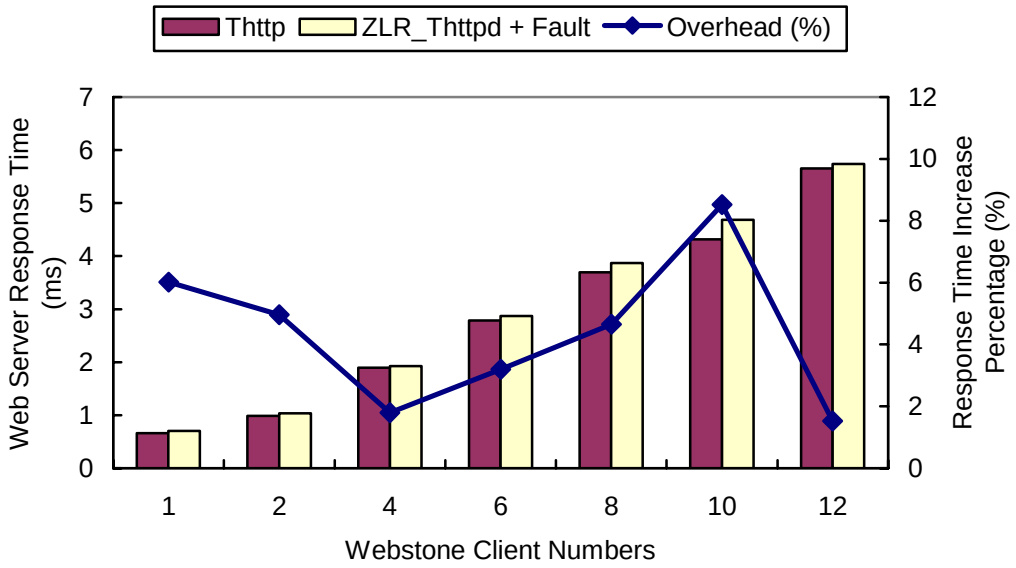


Figure 5.9: Response Time Comparison between Original Thttpd and ZLR_Thttpd with fault.

Figure 5.9 compares the response time between the original thttpd and the ZLR_thttpd with a fault occurs. Similar to the throughput result, there is little difference between the two servers, which ranges from 1.52% to 8.52%.

According to the above two experiments, we can see that our framework has low overhead (i.e., less than 8.52%) when a fault occurs.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 CONCLUSION

In this thesis, we proposed a framework that achieves the goal of zero-loss restart for Internet services through application-kernel cooperation. The framework consists of the logic-state-divided programming style and some required kernel supports. One of the kernel supports is keeping the I/O channel of the faulty Internet service. It has small overhead than reconstruct the communication channel from outside, and it is suitable for Internet service restart. We also provided a kernel support to keep the input data when fault occurs. It is suitable for Internet service which's read frequency is much lower than its write frequency. Finally, we designed an automatic service restart procedure, which contains a fault detection mechanism. The fault detection mechanism can tell the difference between transient fault and general software bug (except software aging problem).

In addition, we deploy our framework on thttpd. The experimental results show that our framework can recover the service from transient faults. Moreover, the runtime and restart overheads are less than 6.8% and 8.5% respectively for small web server, thttpd. It implies that our framework feasible for achieving the goal of non-stop serving.

6.2 FUTURE WORK

The shared memory usage of our framework requires service developers to maintain and adjust the pointers stored in it. To ease the effort of the developers, the

adjustment can be implemented in a user library. Moreover, shared memory regions can automatically be attached by the kernel during restarting phase.

A large-scaled Internet service may consist of several corporative processes with parent-child relationship. Sometimes the service relies on the process relationship to work correctly. In the current implementation, we do not maintain the relationship once a process is failed and then restarted. The relationship maintenance will be integrated into out framework in the future.

REFERENCE

[1] ACME Laboratories, "thttpd - tiny/turbo/throttling HTTP server", available at http://www.acme.com/software/thttpd/.

[2] N. Aghdaie, Y. Tamir, “Client-Transparent Fault-Tolerant Web Service”, 20th IEEE International Performance, Computing, and Communications Conference, Phoenix, AZ, pp. 209-216, April 2001.

[3] L. Alvisi, T.C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov, “Wrapping Server-Side TCP to Mask Connection Failures”, In Proceedings of INFOCOM 2001, pages 329–337, 2001.

[4] Aaron B. Brown, David A. Patterson, “Undo for Operators: Building an Undoable E-mail Store”, USENIX Annual Technical Conference, General Track 2003: 1-14.

[5] George Candea, Armando Fox, “Crash-only software”, In Proc. 9th Workshop on Hot Topics in Operating Systems, Lihue, HI, June 2003.

[6] George Candea, James Cutler, Armando Fox, “Improving Availability with Recursive Microreboots: A Soft-State System Case Study”, Performance Evaluation Journal, Vol. 56, Nos. 1-3, March 2004.

[7] Y. Chawathe, E. A. Brewer, "System Support for Scalable and Fault Tolerant Internet Service", In Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98), Lake District, UK, Sep. 1998.

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

[9] Shang-Te Hsu, Ruei-Chuan Chang, "Continuous Checkpointing: Joining the Checkpointing with Virtual Memory Paging", Software Practices and Experiences, Vol. 27, No. 9, pages 1103-1120, 1997.

[10] Y. Huang, C. Kintala, N. Kolettis, N. D. Fulton, "Software Rejuvenation: Analysis, Module and Applications", in Proceedings of the 25th Symposium on Fault Tolerant Computer Systems, pp. 381-390, Pasadena, CA, June 1995.

[11] C. R. Landau, "The Checkpoint Mechanism in KeyKOS", in Proceedings of the Second International Workshop on Object Orientation in Operating Systems, pp. 86-91, September 1992.

[12] LBNL's Network Research Group, "tcpdump - dump traffic on a network", available at http://www.tcpdump.org/.

[13] C.C.J. Li, W.K. Fuchs, "CATCH-Compiler-Assisted Techniques for Checkpointing", In Proc. of the 20th Annual International Symp. Fault-Tolerant Computing, pages 74-81.

[14] K. Li, J.F. Naughton, J.S. Plank, "Real-time, Concurrent Checkpoint for Parallel Programs", In Proc. 2nd Annual ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, pages 79-88, ACM, March 1990.

[15] J. Long, W.K. Fuchs, J.A. Abraham, "Compiler-Assisted Static Checkpoint insertion", In Proc. of the 22th Annual International Symp. on Fault-Tolerant Computing, pages 58-65, July 1992.

[16] Mindcraft Inc., "Webstone: The Benchmark for Web Servers", available at http://www.mindcraft.com/benchmarks/webstone/.

[17] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, Noah Treuhaft, "Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies", Computer Science Technical Report UCB//CSD-02-1175, March 15, 2002.

[18] Performance Technologies Inc., 2001. The Effects of Network Downtime on Profits and Productivity - A White Paper Analysis on the Importance of Non-stop Networking. White Paper. Available at http://whitepapers.informationweek. com/detail/RES/ 991044232_762.html.

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

[20] James S. Plank, Micah Beck, Gerry Kingsley, Kai Li, "Libckpt: Transparent Checkpointing under Unix", In Proceedings of the Usenix Winter 1995 Technical Conference, pp. 213-223, New Orleans, LA, January, 1995.

[21] M.E. Staknis. "Sheaved Memory: Architectural Support for State Saving and Restoration in Paged Systems", In Proc. of the 3rd International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS), pages 96-103, May 1989.

[22] Florin Sultan, Aniruddha Bohra, Liviu Iftode, "Service Continuations: An Operating System Mechanism for Dynamic Migration of Internet Service Sessions", p177, SRDS'03.

[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 Conference on Distributed Computing Systems, pp. 469-470,

Vienna, July 2002.

[24] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung, Chandra Kintala, "Checkpointing and Its Applications", In Proceedings of the International Symposium on Fault-Tolerant Computing, pp. 22-31, June 1995.

[25] Chu-Sing Yang, Mon-Yen Luo, “Realizing fault resilience in Web-server cluster”, Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), p.21-es, November 04-10, 2000, Dallas, Texas, United States.

[26] Chu-Sing Yang, Mon-Yen Luo, “Constructing zero-loss Web services”, In Proceedings of the 20th IEEE International Conference on Computer Communications (INFOCOM 2001), IEEE Computer Soc. Press, Los Alamitos, CA, 1781--1790.

[27] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. Bressoud, “Engineering Fault-Tolerant TCP/IP Servers Using FT-TCP”, In Proceedings of the 2003 International Conference on Dependable Systems and Networks, San Francisco, California, 22-26 April, 2003.