SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2007; **37**:833–855
Published online 23 November 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.787

Kernel support for zero-loss Internet service restart

Da-Wei Chang[1,*,†], Chuan-Ming Tsai[2], Wei-Kou Li[2] and Tzu-Rung Lee[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

Owing to long serving time and huge numbers of clients, Internet services can easily suffer from transient faults. Although restarting a service can solve this problem, information of the on-line requests will be lost owing to the service restart, which is unacceptable for many commercial or transaction-based services. In this paper, we propose an approach to achieve the goal of zero-loss restart for Internet services. Under this approach, a kernel subsystem is responsible for detecting the transient faults, retaining the I/O channels of the service, and managing the service restart flow. In addition, some straightforward modifications to the service should be made to take advantage of the kernel support. To demonstrate the feasibility of our approach, we implemented the subsystem in the Linux kernel. Moreover, we modified a Web server and a CGI program to take advantage of the kernel support. According to the experimental results, our approach incurs little runtime overhead (i.e. less than 3.2%). When the service crashes, it can be restarted quickly (i.e. within 210 μs) with no information loss. Furthermore, the performance impact due to the service crash is small. These results show that the approach can efficiently achieve the goal of zero-loss restart for Internet services. Copyright © 2006 John Wiley & Sons, Ltd.

Received 12 September 2005; Revised 18 July 2006; Accepted 3 August 2006

KEY WORDS: zero-loss service restart; Internet; transient fault; operating system

1. INTRODUCTION

In recent years, Internet services have become increasingly popular in our daily life. Moreover, the emergence of many transaction-based Internet services such as on-line banking, reservations, and shopping has put a requirement of high availability on such services. According to previous research [1], only a few minutes of downtime can lead to a great financial loss for these services.

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

However, owing to long serving time and huge numbers of clients, Internet services can easily suffer from transient faults. Currently, the main approach to working around these 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 loss 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 regard service application faults as server node failures and thus use heavyweight techniques for recovery, which either lead to a long recovery time [2,3] or require expensive server replicas [4–6]. Others have a requirement that a service should be made up of many fine-grained components, which cannot be satisfied by many existing Internet service programs owing to performance considerations [7–9].

In this paper, we propose a lightweight approach that can achieve the goal of zero-loss Internet service restart. It can solve the problem of transient service faults with little service recovery time and runtime overhead. Moreover, it is cost-effective since it does not require expensive server replicas. It can restart the service on the original machine with no state loss.

The main concept of our approach is to support zero-loss service restart through a kernel subsystem that incorporates the following mechanisms.

- Fault detection. The fault detection mechanism can detect transient faults and then trigger the recovery job.
- I/O channel information retention. The I/O channels that are currently used by the service are maintained and then migrated to the new service instance after the service restart. Maintaining the I/O channels is required since the channels are a part of the service state.
- Restart management. A kernel-level restart manager is responsible for automatically restarting the service when a fault is detected, eliminating the need for restarting the service manually.

To take advantage of the kernel support, some modifications to the service application are needed. Specifically, inspired by the concept of crash-only software [7,8], we require the service to store its state in a long-life, dedicated state storage provided by the kernel, which enables the service state to be safely migrated to the new service instance. In addition, a service should implement a recovery procedure to restore 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 approach to be applied on most existing Internet services.

To demonstrate the feasibility of this approach, we implemented the kernel subsystem as a Linux kernel module, and modified a popular tiny Web server (thttpd [10]) as well as a small CGI program to take advantage of the kernel support. According to the experimental results, our approach incurs less than 3.2% throughput overhead. Moreover, it can restart the service quickly (i.e. within 210 μs) with no information loss when the service crashes. Finally, the performance impact due to the service crash is small. These results demonstrate that the proposed approach can efficiently achieve the goal of zero-loss restart for Internet services.

The rest of the paper is organized as follows. We describe related work in Section 2, followed by a detailed description of the kernel support in Section 3. In Section 4, we describe how we modify thttpd and the CGI program to take advantage of the kernel support. Section 5 shows the performance results. Section 6 discusses the issues in modifying a multi-process Internet server to make it zero-loss restartable. Finally, the paper is concluded in Section 7.

DOI: 10.1002/spe

2. RELATED WORK

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

Checkpointing [11] is one of the most well-known approaches for system recovery. It checkpoints the software state into a stable storage. When a fault occurs, the system can be recovered from the last checkpointed state. This approach can be applied on different levels, such as user library level [12,13], compiler level [14–16], operating system level [17,18], and hardware level [19]. Although this approach can recover a system from transient faults, it is not suitable for service applications that contain hard-to-be-detected bugs, which cause them to crash after a long time of execution. This is because the recovered state is aged, instead of fresh, and thus the service may crash again immediately after the recovery. In this situation, to restart a fresh copy of the service is a more suitable approach. In addition, many checkpoint techniques incur large overheads owing to the large amount of the checkpointed state and the access to the stable storage. Since our approach does not address server node failures but only service faults, we use memory for state storage. Moreover, we can reduce the amount of state that needs to be saved through application–kernel cooperation.

The concept of developing recovery-oriented software for dealing with errors was proposed by the Recovery-Oriented Computing (ROC) project [20], which is a joint effort of University College Berkeley and Stanford University. Different from the previous research, which usually addressed the Mean Time to Failure (MTTF), ROC offered high availability by reducing the Mean Time to Repair (MTTR). In ROC, several techniques related to our method were proposed. Rewind-Repair-and-Replay (3R) [21] 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 had occurred, tries to repair the error, and then replays the operations. However, since we address transient faults, 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) [7,8] is another technique proposed by ROC. Similar to our work, it addresses the transient faults by using the software restart mechanism. Scalable Network Services (SNS) [9] presents an architecture that supports scalable and fault-tolerant Internet services. Both of these approaches require that a service should be made up of fine-grained components. However, many existing Internet service programs cannot satisfy this requirement owing to performance considerations since the inter-component communication will degrade the system performance. In contrast, our approach does not have this requirement and so is more feasible for existing Internet services.

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. In contrast, our approach is totally transparent to the clients. Moreover, Service Continuation only copes with network congestion or server overload conditions and does not deal with server failures.

FT-TCP [2,3] proposed a method to resolve the faults on TCP-based Internet services by inserting wrappers around the TCP layer to log (to a remote machine) the state of the TCP connections associated with the service. If the system crashes, then the TCP connections are re-established, and the service state is reproduced by running a new copy of the service application from the beginning and feeding it

DOI: 10.1002/spe

with the logged I/O requests. That is, it replays the process before the server crashed. The advantage of this approach is that it is transparent to both the server applications and the client side. In addition, it allows the service to survive server node failures by restarting the service and replaying the log on another server node. However, the replaying process may take a long time. Since we address transient faults on the service itself (instead of the server node), and allow the service to cooperate with the framework, we use a more lightweight approach, which eliminates the need to replay the service application and reestablish the TCP connections. This results in much better performance, especially for heavy-loaded Internet services.

Phoenix/APP [24] provides a recovery guarantee for multi-tier component-based applications. In addition to recovering application failures, it can also recover server node failures. The framework intercepts and logs inter-component interactions, which can be replayed during recovery. To minimize the normal execution overhead, the framework tries to reduce the number of forced logs. It also uses the checkpointing technique so that the replaying process can be started from the last checkpointed state before crash, reducing the replaying time. However, checkpointing causes synchronous I/O and thus makes this technique costly.

There are also some other research efforts that address fault-tolerant Web services [4–6]. They use the server replication approach to improve the availability of the Web service. 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. In contrast, we address the transient faults occurring on the service itself, and thus 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 them over to the new service generation to help achieve the goal of zero-loss service restart.

3. KERNEL SUPPORT FOR ZERO-LOSS SERVICE RESTART

As shown in Figure 1, the life cycle of a restartable service consists of four kinds of phase. The first kind is the *service initialization* phase, in which the service initializes its serving environment and starts waiting for requests. The second is the *service execution* phase. In this phase, the service processes incoming requests until a fault occurs. The third is the *process restarting* phase, in which the kernel detects the service failure and restarts a new service generation with the same I/O channels. Finally, the fourth is the *service recovery* phase. In this phase, the restarted service restores its state information. After the service recovery phase, the restarted service again enters the service execution phase, which allows it to continue serving the on-processing requests and to start accepting new requests.

Existing APIs exported by operating systems have some limitations to achieve the goal of zero-loss service restart. First, the detection and restarting of a failed process are not fully automatic because the administrator must manually restart a new process. Second, no matter whether a process exits normally or abnormally, its open files will be closed by the kernel. This will break all channels between the 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 recover. This is true for non-storage-based files, such as pipes, sockets, etc. When a read operation 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 service restart, some additional kernel support is needed.

DOI: 10.1002/spe

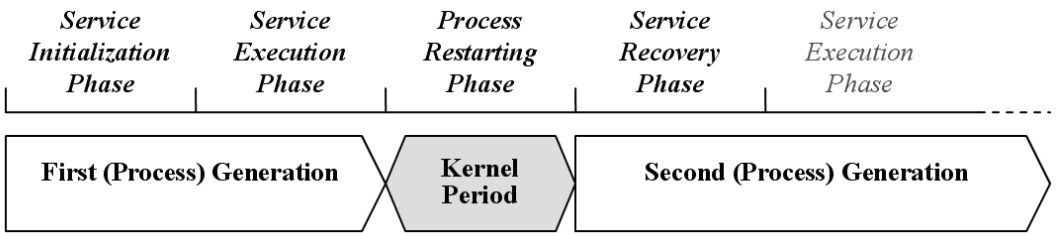


Figure 1. Life cycle of a restartable service.

In this paper, we design a kernel subsystem that incorporates three mechanisms to support zero-loss service restart. These mechanisms are fault detection, I/O information retention, and restart management. In the following, we describe the design and implementation of these mechanisms in the Linux kernel.

3.1. Fault detection

When a fault causes a service to crash, the operating system will terminate the service process and release the resources owned by the service. 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 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). Therefore, we can simply check the bit to see if the process termination is abnormal.

However, non-transient faults can also cause abnormal process termination, and such faults may occur again soon after the service restart. To avoid this problem, we add a per-process variable, *last_restart_time*, to record the last time at which the service process was restarted. When a fault is detected, we consider the fault as transient if the difference between the current time and the *last_restart_time* of the service process is larger than a certain threshold (currently, 1 minute). We only restart the service for transient faults since non-transient faults cannot be recovered by service restart.

Note that the fault detection mechanism does not trigger the service restart flow for all processes. Instead, a service has to register itself to the kernel if it wants to be zero-loss restartable. When a fault occurs in a restartable process, the kernel uses the registration information to create a new generation of the process and then performs the recovery job. The registration is done through the *reregist()* system call, which is shown in Figure 2.

The first three parameters of this system call represent the path and arguments of the service program. 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 actions for recovery. By registering the role of the current process, the restarted process can perform the recovery actions accordingly. For example, Apache server version 1.3 consists of one master process and a number of worker processes. The former controls the number of the latter, which is actually responsible for request processing. In order to become restartable, the master process can register itself with the *child_id* as 0

DOI: 10.1002/spe

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

Figure 2. Prototype of the restart registration system call.

and all the workers can register themselves with *child_id* as 1. As a consequence, if a process crashes, the restarted process can determine whether it should continue to control the number of the workers or it should continue serving the on-processing request according to the value of *child_id*.

In addition, we also provide a *getreginfo()* system call so that the service process can retrieve its registration information. This is primarily used by the next service generation to get the registration information of its previous generation.

3.2. Retaining 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, but rather they should be kept and handed over to the next service generation. Keeping I/O channels consists of three tasks: preventing the I/O channels from being closed; retaining the unprocessed input data; and logging the progress of the output channels. We describe these tasks in the following subsections.

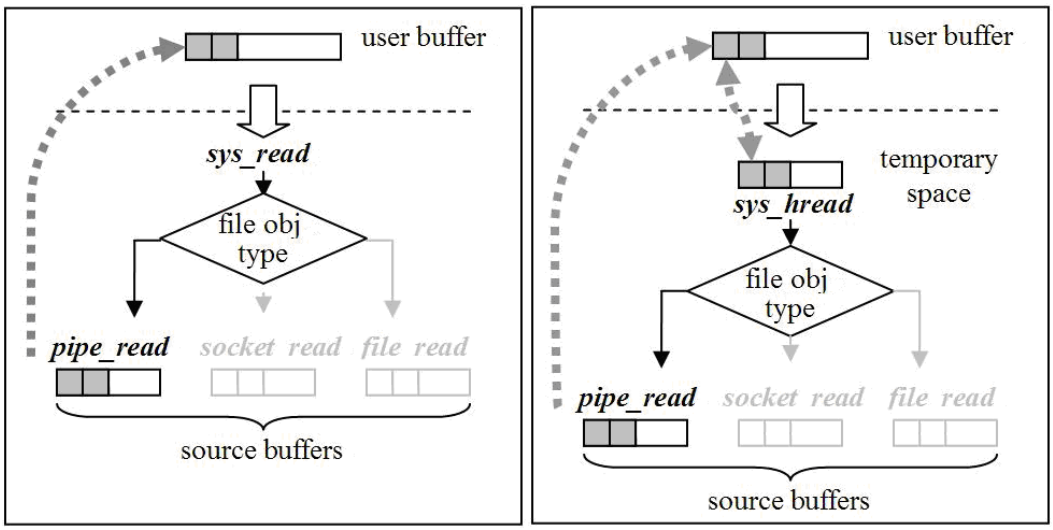
3.2.1. I/O channel retention

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 open files of this process. Therefore, we can retain the I/O channels of a service process by preventing all the files in the process's open file table from being closed. A simple approach to achieve this is to increase the reference count of the open file table. As a result, the kernel will not close the files and free the 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. Holding data in input channels

As mentioned above, a process failure will cause the loss of all data that has been read into the memory space of the process but not yet been handled. To avoid this problem, we propose an alternative read operation, namely data-holding read. Figure 3 shows a comparison between original read and data-holding read operations. From this 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 a pipe

DOI: 10.1002/spe



(a) (b)

Figure 3. Original read versus data-holding read.

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.

Different from the original read operation, data is removed *explicitly* in the data-holding read. That is, the user program can specify not only the number of bytes to read but also the number of bytes to delete. We implement the data-holding read operation by maintaining 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 that has already been read to the user buffer but not yet been specified to be deleted. Thus, the already-read data can be retained in the temporary space until the service does not need it.

Figure 4 shows the prototype of the 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. The behavior of the system call is as follows. First, it tries to return at most *rlen* bytes of data directly from the temporary space. If the data in the temporary space is not enough, the remaining data bytes will be read by issuing an original read operation that copies the data from the kernel-level source buffer to the user buffer. These bytes will then be copied to the temporary space (from the user buffer) to keep them in the kernel. Note that, during the execution of the system call, *dlen* will be checked to see if the user wants to delete more data than he/she has read.

DOI: 10.1002/spe

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

Figure 4. Prototype of data-holding read system call.

```
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 5. Definition of the *dhr* structure.

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 service program invokes *hread()* on that file for the first time. Figure 5 lists the fields of the *dhr* structure and the related data structures.

Each *dhr* structure has a pointer to the corresponding file, and a pointer to a chain of fixed-size buffers (i.e. *dhr_buf*) for holding the already-read data. In addition, it uses three pointers to manage the buffer chain, as shown in Figure 6. The *read_pt* and *destroy_pt* point to the last byte of data that has been read and deleted by the user process, respectively. Therefore, the data between the two pointers represent the bytes that have already been read but have not been deleted by the user program. The *write_pt* points to the last byte of data that has been copied into the buffer chain. Note that the values of the *read_pt* and *write_pt* may be different, which happens when a user specifies an invalid *dlen* parameter in the *hread()* system call. In this situation, the kernel clears the user buffer and returns an error message to the user. However, it still copies the data into the kernel temporary space (before the cleaning of the user buffer) and updates the *write_pt* accordingly.

3.2.3. Logging progress of output channels

After obtaining the I/O channels, the restarted service should continue sending data to its output channels. To achieve this, it must know how many bytes it had written to each of its output channels

DOI: 10.1002/spe

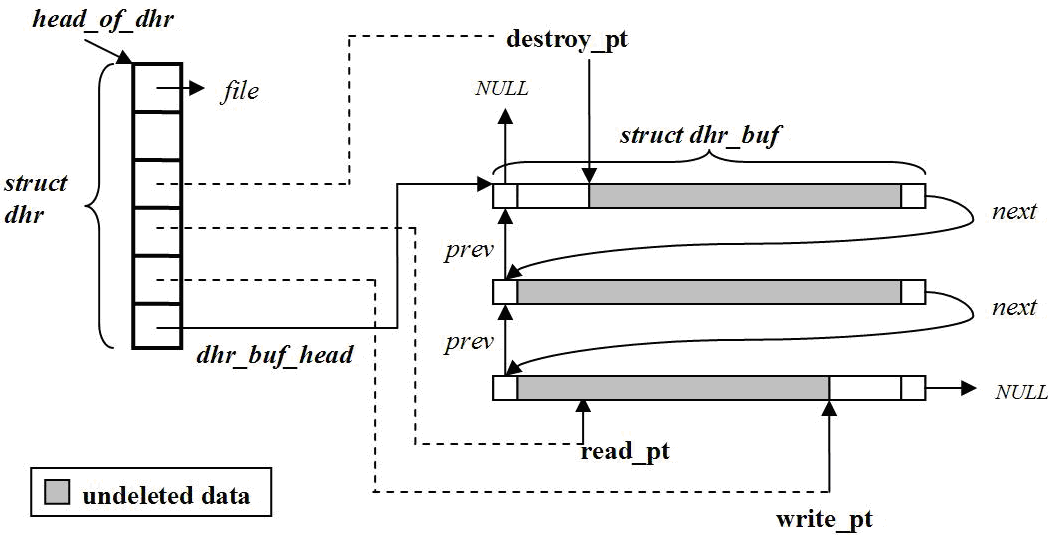


Figure 6. Structure *dhr* and structure *dhr_buf*.

when the failure happened. A service program can record this information in a long-lived state storage. However, such information recording cannot be done together with the write operation in an atomic way. This causes problems when a failure happens between the write operation and the information recording. Specifically, the restarted service may skip some data or resend some already-been-sent data to the clients, causing errors on the clients.

To solve this problem, we provide an alternative write operation named *lwrite()*, which (in addition to perform the original write operation) logs the number of bytes the file has been written by the service program. During the service restart, the service program can obtain the logged information from the kernel so that it can learn how many bytes have been sent for each of its output channel.

3.3. Restart flow

The restart management mechanism controls the service restart flow, as shown in Figure 7.

First, a fault in the service process is detected by the fault detection mechanism. Then, the kernel performs the I/O channel retention 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* (mentioned in Section 3.2.2) in the kernel temporary space to the value of the *destroy_pt* so that the unprocessed data can be read again by the new service generation.

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

DOI: 10.1002/spe

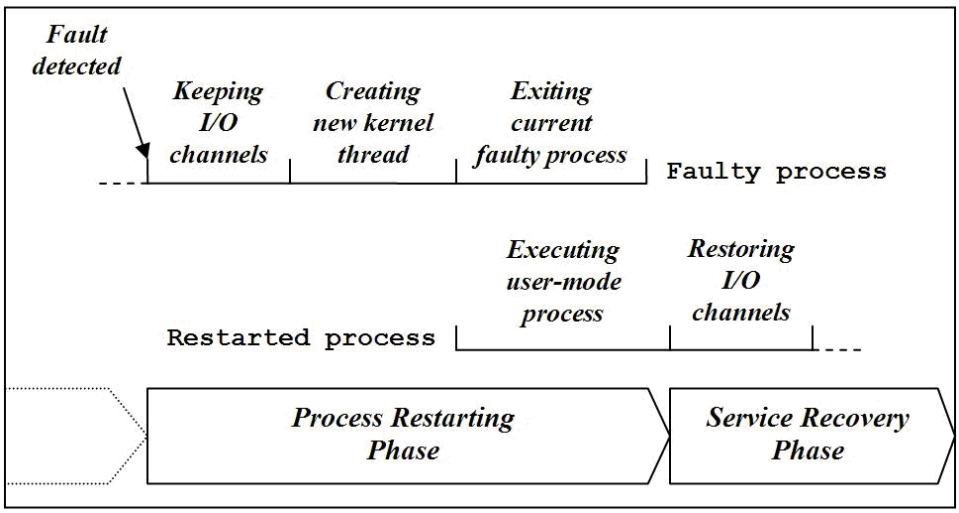


Figure 7. Service restart flow.

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 refers to the open file table into the task control block of the new service generation.

4. SERVICE SIDE COOPERATION

In order to achieve the goal of zero-loss service restart, a service should be modified to take advantage of the above kernel support. First, it should register itself to the kernel, and replace the *read()/write()* system calls with *hread()/lwrite()*. Second, it should store the service state in the kernel-provided state storage. In our system, the storage is implemented via shared memory, which remains alive even when the process crashes. Third, the service should implement a recovery procedure for restoring its state according to the content of the state storage. As shown in Figure 8, a service program determines whether or not it is restarted, via the *getreginfo()* system call, during the program initialization. If it is restarted, the program executes the recovery procedure according to the returned *child_id*. Otherwise, it follows the normal path.

In the following, we will describe the modifications to a popular tiny Web server thttpd [1] and a small CGI program to make them zero-loss restartable.

DOI: 10.1002/spe

```
#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) ) {                    // Initialization
     is_restarted = reg_info->is_restarted;

     }

     if(is_restarted ) {                       //  recovery path

         switch(reg_info->child_id) {

               case 0: goto recover_parent;

               case 1: goto recover_child1;

               case 2: goto recover_child2;

               ...

         }

     }

     else  {                                     // normal execution path

     ...

     }

}
```

Figure 8. Execution paths of a zero-loss restartable service.

4.1. Zero-loss restartable thttpd

The single-process Web server thttpd implements HTTP 1.1 [25]. It handles concurrent requests by using the *select()* system call. As shown in Figure 9, thttpd divides the request handling into two stages: *reading* and *sending*. In the *reading* stage, thttpd 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.

DOI: 10.1002/spe

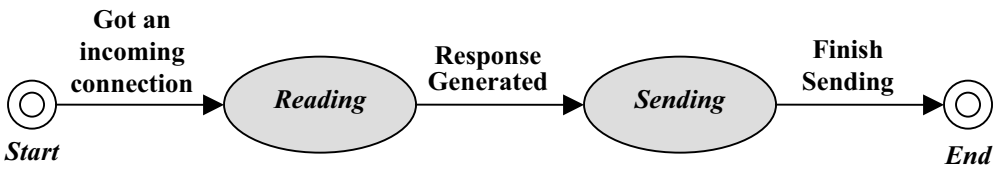


Figure 9. State flow of request handing in thttpd.

We made the following modifications to thttpd in order to take advantage of the kernel support so that it can be restarted with no information loss. The modified version is called ZLR_thttpd. First, we replaced all socket read operations in thttpd with *hread()*. Before completing the parsing of a request, the request is stored in the kernel-level temporary space so that it will not be lost owing to a server crash. Second, we replaced all socket write operations with *lwrite()* in order to log the progress of each output channel. Therefore, the progress of each output channel can be retrieved from the kernel during the service restart. Third, we applied the *reregist()* and the *getreginfo()* system calls as mentioned in Section 3.1 and Figure 8. Fourth, we identified the state variables of thttpd and stored them in shared memory areas (i.e. the state storage) so that we can recover the state of thttpd when it is restarted. For a Web server such as thttpd, the most important state is the state of the on-processing connections. In thttpd, each *connection entity* structure is allocated to store the information of an on-processing connection. Therefore, the state variables of a connection can be obtained by extracting from this structure the fields that are necessary for the recovery procedure.

Figure 10 shows the state variables of the ZLR_thttpd. The per-connection state variables are extracted from the connection entity structure and are grouped into another structure called *http_state_var*, which is shown in the left-hand part of the figure. The fields in this structure can be divided into three parts. The first part contains the *conn_stage* and *conn_fd* fields. The former represents the processing stage of this connection (i.e. *reading* or *sending*), and the latter represents the data socket used for communicating with the client. The second part is updated during the *reading* stage of a connection. The *expnfilename* field represents the requested file name, 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 indicates how many bytes of response have been written to the data socket. In addition to the per-connection state variables, the global state variables are maintained in the *global_state_var* structure, as shown in the right-hand part of Figure 10. The *listen_fd* represents the socket that the server uses to accept connections 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.

Note that, instead of separating the original data structures in thttpd into state-variable part and non-state-variable parts, we make a copy of all state variables and store the copy into the shared memory. In addition, we update a variable in the shared memory before the corresponding variable of thttpd is modified. These can avoid large modifications to the original thttpd program.

DOI: 10.1002/spe

```
struct httpd_state_var {
        int conn_stage;
        int conn_fd;                    ①

        char* expnfilename;
        int method;                     ②

        off_t bytes_sent;               ③
};

struct global_state_var {
        int listen_fd;
        int num_connects;

        int max_connects;
};
```

Figure 10. State variables of ZLR_thttpd.

As shown in Figure 11, we used four shared memory areas to store the above state variables. Areas A and B are used to store the per-connection variables[‡], area C stores the global variables, and area D records the attached address of area B. The attached address is recorded because the data in area B will be referenced by data in the other areas such as area A, and area B may be re-attached to another address of the restarted process. If area B is re-attached to a different address, the *expnfilename* field of area A should be adjusted according to the difference between the attached and the re-attached addresses. Therefore, we used an extra area to record the attached address of area B[§].

The last modification to thttpd is that we added a recovery procedure, which will be executed during the service recovery phase. The procedure is straightforward. First, it re-attaches the shared memory areas. Second, it re-constructs the internal data structures according to the content in those areas. For example, it re-constructs the *connection entity* structure for each on-line connection according to the state variables (i.e. the *httpd_state_var* structure shown in Figure 10) stored in the share memory areas. Third, it rolls back the processing of each on-line request to a proper state. As shown in Figure 12, the processing of a request can be divided into several fine-grained states. In the figure, the *normal* lines indicate the normal processing flow. After a new connection is accepted, the Web server reads the request, parses the request, and records the *expnfilename*. Then, it maps the requested file into its address space and starts sending out the file content. Note that each request has a corresponding state machine as shown in Figure 12 and the state transition happens whenever the server finishes one of the above jobs. For example, the request enters into state *request_parsed* when the parsing of the request is finished.

The *rollback* lines indicate the state rollback that happens whenever the server restarts. For example, if a request was in state *request_parsed* when the server crashed, it will be rolled back to state *connection_accepted* so that the server can read the request again. This is because, although the request has been parsed, the result (i.e. the *expnfilename*) has not been recorded. From Figure 12 we can see that, in the first five states, if the process crashes before the *expnfilename* is recorded, the request will be read again from the kernel temporary space; otherwise, we only have to go back

[‡]We put the variable-sized field of the *httpd_state_var* structure (i.e. *expnfilename*) into another area for easy maintenance.
[§]In the future, we will ensure that a shared memory area will always be attached to the same address by different service generations so that there is no need for such pointer adjustment and extra shared memory areas.

DOI: 10.1002/spe

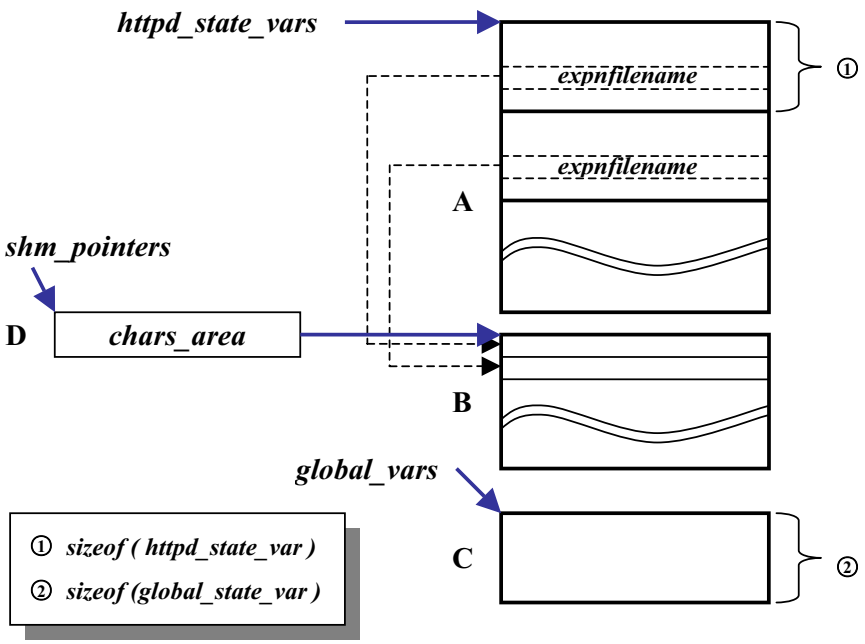


Figure 11. Four shared memory areas in ZLR_thttpd.

to state *expnfilename_recorded* and re-map the requested file into the server's memory space. In state *some_response_sent*, the restarted server goes back to state *expnfilename_recorded* so that it can map the file into its address space again and continue sending the remaining bytes to the client. In state *response_sending_completed*, the request remains on the same state. This causes the server to terminate the connection since all the bytes were sent to the client. When the processing of each request has been rolled back to a proper state, the server can continue processing the on-line requests and accepting new requests.

4.2. Zero-loss restartable CGI

In this section, we describe how to modify a CGI program, which involves dynamic content delivery and user session states, to make it zero-loss restartable. To make this section easier to understand, we describe issues about dynamic content delivery issues first, and then give discussions related to session states.

4.2.1. Dynamic content delivery

We constructed a CGI program with the following behavior. Each time the CGI program is invoked, it randomly selects one of the text files stored on the server, and then it sends out the name of the file, the content of the file, and the current access time, to the client. Simply restarting the service program and

DOI: 10.1002/spe

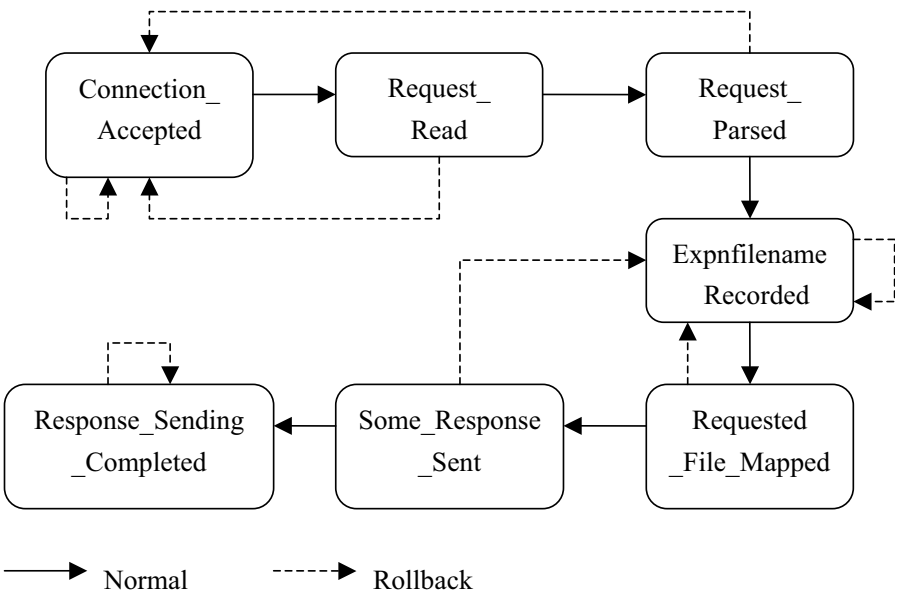


Figure 12. State rollback of the ZLR_thttpd.

replaying the request cannot recover such a CGI program seamlessly. This is because the data sent to the client may be inconsistent before and after the service failure.

To make the CGI program zero-loss restartable and maintain the server output consistency, we performed the following modifications to the CGI program. First, we used a shared memory area to store the non-deterministic part of the CGI output (i.e. the access time and the name of the accessed file) so that the output of the restarted CGI program will be consistent with that of its previous generation. Second, we used the *hread()* and *lwrite()* systems calls to retain the unhandled input data and to record the number of bytes that have been sent to the client. Third, we applied the *reregist()* and *getreginfo()* system calls to register the CGI program to the framework and to check if the CGI program is restarted or not.

4.2.2. Session states

Session states contain information about a specific user. Generally, a session starts when the user logs into the system, and it ends when the user logs out or remains idle for a long period of time. To demonstrate session states, we modified the CGI program mentioned above to make it also send to the client a name list of files that have been accessed during the user session.

Since session states should be shared among CGI programs that are invoked by the same user, we support zero-loss session states by storing them into per-session shared memory areas. Each time a user logs into the Web site, a session with a unique session identifier (SID) starts, and the corresponding

DOI: 10.1002/spe

shared memory area is created. We use SID as the key of the shared memory area so that every CGI programs corresponding to the same user can attach the same shared memory area by providing the SID. For each CGI program, the SID is stored in another memory area that is shared between the generations of the CGI program. Therefore, a restarted CGI program can obtain the SID and then attach the shared memory area that stores the session states.

In addition to session states, global states that are shared among all CGI programs can be implemented in a similar way. The states can be stored in a shared memory area with a well-known key and thus all CGI programs can access the states.

To achieve zero-loss recovery of the session/global states, we apply an approach similar to Write-Ahead Logging (WAL) when we modify the states. We provide a shadow copy of the states on the shared memory area. Before updating the states, a service application acquires the semaphore and writes the target values of the states to the shadow copy first, and then it starts updating the states. If the application crashes before all the states are completely updated, the restarted instance can learn that by comparing the values of the states with those of the shadow copy, and continue updating the states. When all the states are updated, the application releases the semaphore. This approach requires that a restarted service can still acquire the semaphore when its previous generation crashed while holding the semaphore. We have modified the *semop()* system call to achieve this. To reduce the burden of the service applications, we will implement a system call for service applications to update the states in an atomic way in the future.

5. PERFORMANCE EVALUATION

In this section, we evaluate the effectiveness and efficiency of the proposed kernel mechanisms. We first compare the performance of the ZLR_thttpd with that of the original thttpd when no fault occurs. Then, we verify the functionality of our restart mechanism by injecting faults to the ZLR_thttpd and the CGI program. We also measure the performance of the ZLR_thttpd in the presence of a fault. The performance is measured by using the WebStone [26] benchmark version 2.5 with the standard testbed profile.

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

5.1. Overhead

Figure 13 shows the throughput comparison between the original thttpd and the ZLR_thttpd when no fault occurs. The x-axis represents the numbers of WebStone clients, each of whom establishes a large number of connections with the server during the experiment. The y-axis indicates the server throughput numbers reported by WebStone. From the figure we can see that using the kernel mechanisms results in only little throughput degradation, ranging from 0.6% to 3.2%, indicating that these mechanisms are quite efficient. The overhead is due to the storing of the state variables to the shared memory and the use of *hread()/lwrite()* for socket reading/writing.

DOI: 10.1002/spe

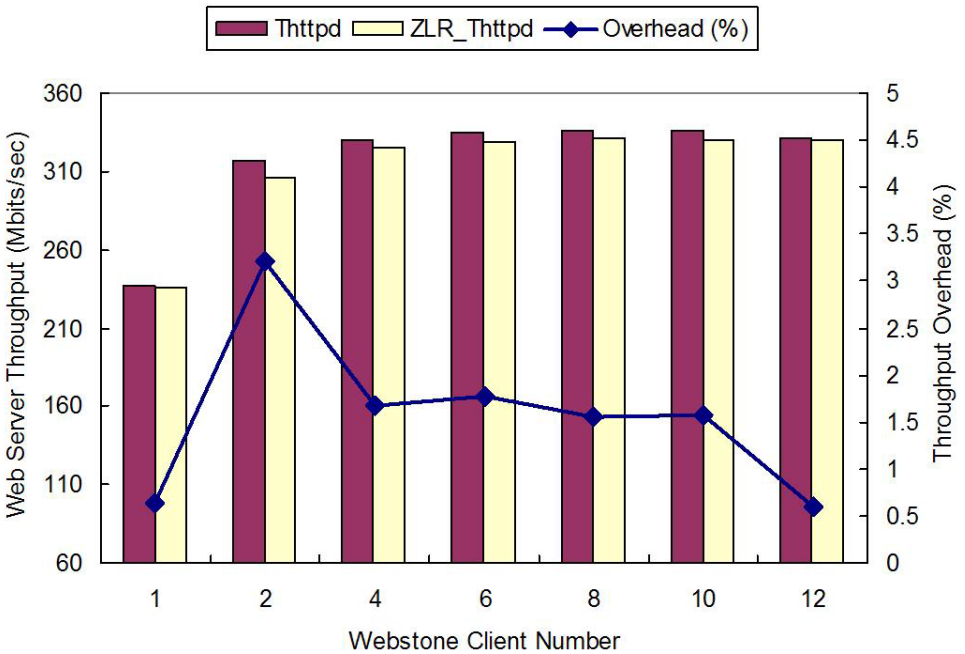


Figure 13. Throughput comparison between thttpd and ZLR_thttpd.

In addition to the throughput overhead, we also measure the space overhead caused by the in-kernel temporary space. Figure 14 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. Performance of service restart

In this section, we first verify the functionality of the restart mechanism. Then, we measure the time needed to restart a ZLR_thttpd service and the performance degradation when a fault occurs.

In this experiment, we show that the ZLR_thttpd can continue serving an online client even when a fault occurs. We make the client issue a connection and request a 5 Mbyte Web page from the server. During the experiment, we inject a fault by sending a segment violation signal to the server, which triggers the restart mechanism.

Figure 15 shows the result. The fault is injected after the server has sent 3 000 000 bytes of the response. As shown in the figure, the server can continue serving (with a new generation) after a fault occurs in it. Although it is not shown in the figure, the data received by the client are exactly the same as the data stored in the server.

DOI: 10.1002/spe

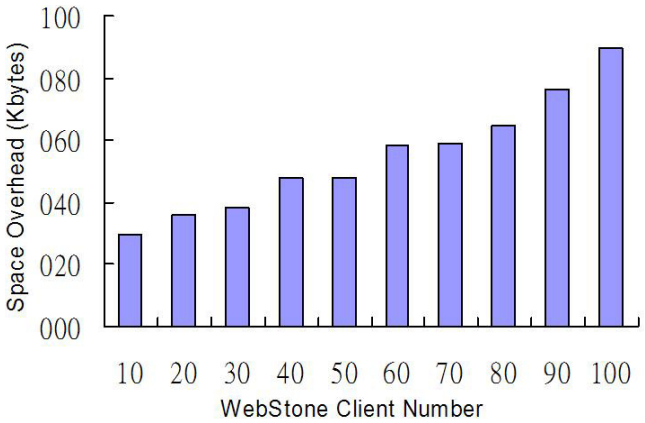


Figure 14. Space overhead of the kernel temporary space.

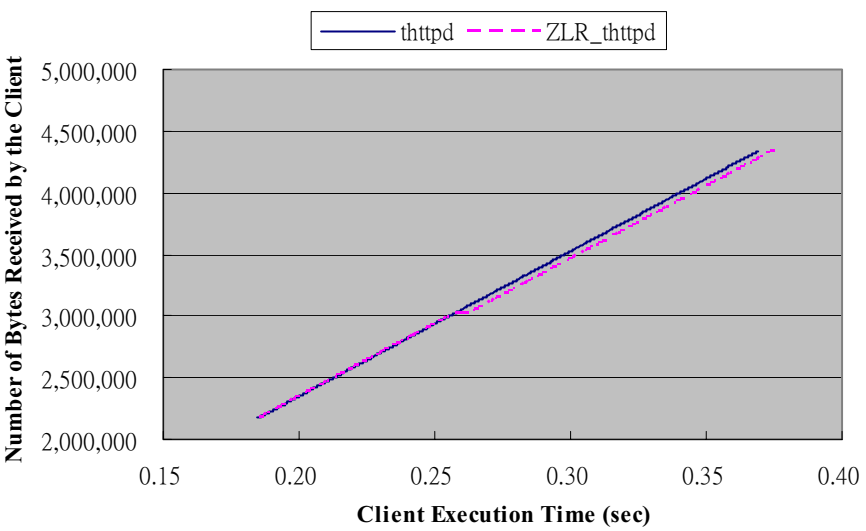


Figure 15. Effect on the client side when a fault occurs in ZLR_thttpd.

DOI: 10.1002/spe

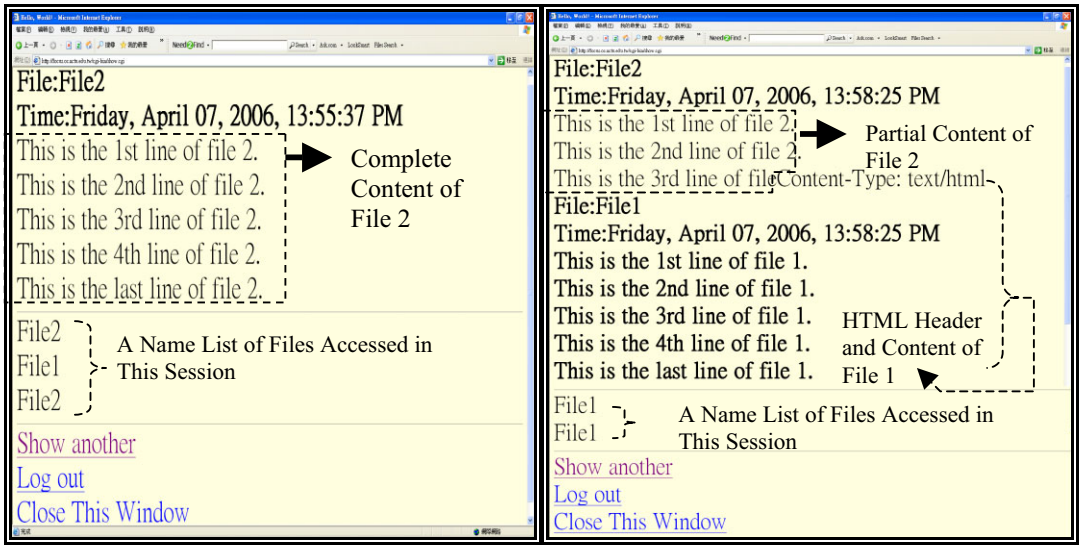


Figure 16. Zero-loss restart (left) versus simple restart (right) on a CGI fault.

Figure 16 compares the result of the zero-loss restart mechanism with that of a simple restart one when a fault occurs in the CGI program[¶]. As mentioned in Section 4.2.1, on each invocation, the CGI program sends out a randomly-selected file to the client. We insert the same segment violation fault during the transfer of the file content. As shown in the figure, if the simple restart mechanism is used, the user may receive an inconsistent result when a fault occurs during the processing of the request. Specifically, the figure illustrates that the fault occurs during the transfer of file2 and the restarted CGI replays the request by sending file1 to the client. In contrast, zero-loss restart allows the CGI program to keep its output consistent.

Table I presents a breakdown of the kernel execution time spent for restarting a new ZLR_thttpd generation. The time begins when the fault is detected and ends when the I/O channels are restored. From the table we can see that the total service restart time is 208 μs. Note that keeping and restoring I/O channels is quite efficient (specifically, only about 7.4 μs). Most of the time is spent on creating the kernel thread and executing the binary image of the new service generation.

Figure 17 shows the throughput comparison between the thttpd and the ZLR_thttpd when a fault occurs, indicating that ZLR_thttpd still results in little throughput degradation. The overhead ranges

[¶]Under the simple restart mechanism, a restarted CGI just replays the request from the beginning. However, we still retain the IO channels for the CGI process so that it can send data to the client via the same network connection as its previous generation. Thus, we can check the output consistency by using a Web browser.

DOI: 10.1002/spe

Table I. Kernel execution time for restarting a new ZLR_thttpd generation.

	Execution time (μs)
Retaining I/O channels	7.30
Creating kernel thread	15.26
Executing user-mode process	185.78
Restoring I/O channels	0.12

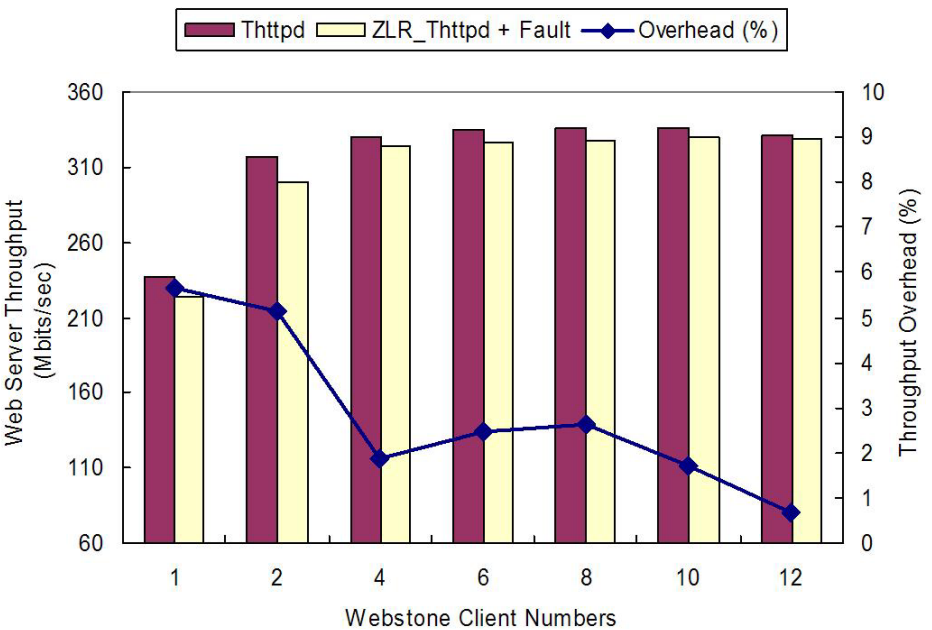


Figure 17. Throughput comparison between thttpd and ZLR_thttpd with the presence of a fault.

from 0.68% to 5.67%, which is similar to the results shown in Figure 13. This shows that the fast restart and the I/O channel retention ensure that there is little performance impact due to the raising of faults.

In summary, the mechanisms in the kernel subsystem lead to small throughput and space overheads. When a fault occurs, the service can be restarted with no information loss, and the performance impact due to the occurrence of the fault is slight. These results show that the proposed approach can efficiently achieve the goal of zero-loss Internet service restart.

DOI: 10.1002/spe

6. MODIFICATIONS TO MULTI-THREADED SERVICES

In Section 4, we have used single-process and single-threaded services as examples to show how to perform the modifications to take advantage of the kernel support. However, the proposed approach does support multi-process and multi-threaded services. Generally, we can perform similar modifications to such services to make them zero-loss restartable. In the following subsections, we discuss some further issues that are worth addressing when modifying a multi-process/multi-threaded service.

6.1. Inter-process communication channels

The major difference between a single-process service and a multi-process service is that the latter uses inter-process communication (IPC) channels for information exchange. For example, a service may consist of one *dispatcher* process and a number of *worker* processes. The former receives requests from the clients and hands the requests to the latter via the IPC channels. Zero-loss service restart requires that the channels should not be broken when a process crashes. We can satisfy this requirement because processes usually use file-based communication channels such as sockets and pipes (including named pipes), and the set of the open file descriptors is retained and then handed over to the next process generation by our mechanisms.

In addition to the file-based communication channels, processes may also communicate via shared memory areas. As we mentioned before, shared memory areas will not be deleted when the process that uses them crashes.

In summary, with the help of the kernel mechanisms we proposed, a process crash will not cause the IPC channels to be broken.

6.2. Different roles for different processes

Different processes of an Internet service may play different roles, and hence require different actions for recovery. For example, Apache server version 1.3 is made up of one *master* process and a number of *worker* processes. The former controls the number of the latter, which actually handle the HTTP requests. A shared memory area called *scoreboard* is used for information exchange among the processes. If the master process crashes, the recovery action taken by the new generation of the master process is to re-attach the scoreboard so that it can obtain the information (e.g. process identifier) of its worker processes. A worker process usually stores the status of the requests it handles into its memory space. To make it zero-loss restartable, we have to modify the worker so that it stores the request status into another shared memory area, say *request_info*. When the new generation of the worker process is started, it should re-attach both the *scoreboard* and the *request_info* areas. The former allows it to communicate with the master process while the latter allows it to continue handling the request.

As we mentioned above, in order to determine the role of a restarted process, we provide a *reregister()* system call for processes to specify their roles. For example, the master process of the Apache server version 1.3 can invoke the system call with the *child_id* parameter as 0, while each worker process can invoke the system call with the *child_id* parameter as 1. If a process is restarted, the *getreginfo()* system call will return the *child_id* of its previous generation. If the returned *child_id* is 0, the restarted process will perform the recovery procedure of the master process. Otherwise, it will perform the recovery procedure of the worker process.

DOI: 10.1002/spe

6.3. Single thread versus multiple threads

If a service is made up of multiple single-threaded processes (e.g., Apache version 1.3 and the default model of Apache version 2.0), the recovery action is the same as we describe above. That is, when a process is restarted, it should obtain the status of its pending requests from the shared memory areas, roll back the processing state when necessary, and then continue handling the requests. However, if the service contains one or more multi-threaded processes, some additional tasks need to be done. First, when the process is restarted, the recovery procedure should create a set of threads according to the number of the pending requests. For example, if the original design of the service uses a thread for each request (this is the typical case) and there are N pending requests, the recovery procedure should create N threads and assign a request to each thread. Second, the entry function of each thread should not process the corresponding request from the beginning. Instead, it should start processing from the current state of the request.

7. CONCLUSIONS

In this paper, we design and implement a kernel subsystem that supports zero-loss restart for Internet services. The subsystem contains a fault detection mechanism that can detect transient faults. Moreover, it retains the I/O channels and the unprocessed input data for the service so that the information will not be lost owing to service crashes. Finally, it incorporates an automatic service restart procedure, which can restart a new service generation and hand over the retained I/O channels. This allows the new service generation to continue handling the requests that were on-line when the service crashed.

In order to achieve the goal of zero-loss service restart, an Internet service should be modified to take advantage of the kernel support. The modification is straightforward and hence it can be applied on most existing Internet services.

We implemented the kernel subsystem as a Linux kernel module and modified a popular tiny Web server thttpd and a CGI program to take advantage of the kernel support. Performance results show that the mechanisms in the kernel subsystem lead to less than 3.2% throughput overhead. When a fault occurs, the service can quickly be restarted (within about 210 μs) with no information loss, and hence the performance impact due to the occurrence of the fault is slight. This demonstrates that the system can efficiently achieve the goal of zero-loss Internet service restart.

Following this current work on the problem of Internet application failures, in the future we will investigate the possibility of incorporating our technique with others to make an Internet server survive after other types of failure. For example, hardware failures can be masked by device redundancy, and some operating system failures can be recovered by techniques such as Nooks [27,28].

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*. Available at: http://whitepapers.informationweek.com/detail/RES/991044232_762.html [2001].
2. Alvisi L, Bressoud TC, El-Khashab A, Marzullo K, Zagorodnov D. Wrapping server-side TCP to mask connection failures. *Proceedings of INFOCOM 2001*, Anchorage, AK, 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001; 329–337.
3. Zagorodnov D, Marzullo K, Alvisi L, Bressoud T. Engineering fault-tolerant TCP/IP servers using FT-TCP. *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, San Francisco, CA, 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003; 22–26.

DOI: 10.1002/spe

4. Aghdaie N, Tamir Y. Client-transparent fault-tolerant Web service. *Proceedings of the 20th IEEE International Performance, Computing, and Communications Conference*, Phoenix, AZ, 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001; 209–216.
5. Yang CS, Luo MY. Realizing fault resilience in Web-server cluster. *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, Dallas, TX, 2000. IEEE Computer Society Press: Los Alamitos, CA, 2000; p.21–es.
6. Yang CS, Luo MY. Constructing zero-loss Web services. *Proceedings of IEEE INFOCOM 2001*, Anchorage, AK, 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001; 1781–1790.
7. Candea G, Fox A. Crash-only software. *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003; 67–72.
8. 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.
9. Chawathe Y, Brewer EA. System support for scalable and fault tolerant Internet service. *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*. Springer: New York, 1998.
10. ACME Laboratories. Thttpd—Tiny/Turbo/Throttling HTTP Server. http://www.acme.com/software/thttpd/ [2006].
11. Plank JS. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. *Technical Report UTCS-97-372*, 1997.
12. Plank JS, Beck M, Kingsley G, Li K. Libckpt: Transparent checkpointing under UNIX. *Proceedings of the USENIX Winter 1995 Technical Conference*, New Orleans, LA. USENIX Association: Berkeley, CA, 1995; 213–223.
13. Wang YM, Huang Y, Vo KP, Chung PY, Kintala C. Checkpointing and its applications. *Proceedings of the International Symposium on Fault-Tolerant Computing*, Pasadena, CA, 1995. IEEE Computer Society Press: Los Alamitos, CA, 1995; 22–31.
14. Li CCJ, Fuchs WK. CATCH-compiler-assisted techniques for checkpointing. *Proceedings of the 20th Annual International Symposium on Fault-Tolerant Computing*, Newcastle, U.K., 1990. IEEE Computer Society Press: Los Alamitos, CA, 1990; 74–81.
15. Li K, Naughton JF, Plank JS. Real-time, concurrent checkpoint for parallel programs. *Proceedings of the 2nd Annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, 1990. ACM Press: New York, 1990; 79–88.
16. Long J, Fuchs WK, Abraham JA. Compiler-assisted static checkpoint insertion. *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, Boston, MA, 1992. IEEE Computer Society Press: Los Alamitos, CA, 1992; 58–65.
17. Hsu ST, Chang RC. Continuous checkpointing: Joining the checkpointing with virtual memory paging. *Software: Practice and Experience* 1997; **27**(9):1103–1120.
18. Landau CR. The checkpoint mechanism in KeyKOS. *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, Paris, France, 1992. IEEE Computer Society Press: Los Alamitos, CA, 1992; 86–91.
19. Staknis ME. Sheaved memory: Architectural support for state saving and restoration in paged systems. *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, Boston, MA, 1989. ACM Press: New York, 1989; 96–103.
20. Patterson D *et al*. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. *Computer Science Technical Report UCB//CSD-02-1175*, 2002.
21. Brown AB, Patterson DA. Undo for operators: Building an undoable e-mail store. *Proceedings of the USENIX Annual Technical Conference*, San Antonio, TX, 2003. USENIX Association: Berkeley, CA, 2003; 1–14.
22. Sultan F, Bohra A, Iftode L. Service continuations: An operating system mechanism for dynamic migration of Internet service sessions. *Proceedings of the Symposium on Reliable Distributed Systems*, Florence, Italy, 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003; 177–188.
23. 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*, Vienna, Austria, 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002; 469–470.
24. Barga R, Chen S, Lomet D. Improving logging and recovery performance in Phoenix/App. *Proceedings of the International Conference on Data Engineering*, Boston, MA, 2004. IEEE Computer Society Press: Los Alamitos, CA, 2004; 486–497.
25. Fielding R, Gettys J, Mogul J, Frystyk H, Masinter L, Leach P, Berners-Lee T. Hypertext Transfer Protocol—HTTP/1.1. *RFC 2616*, 1999.
26. Mindcraft Inc. WebStone: The benchmark for Web servers. http://www.mindcraft.com/benchmarks/webstone/ [2005].
27. Swift M, Annamalai M, Bershad BN, Levy HM. Recovering device drivers. *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, 2004. USENIX Assocation: Berkeley, CA, 2004.
28. Swift M, Bershad BN, Levy HM. Improving the reliability of commodity operating systems. *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, 2003. ACM Press: New York, 2003.

DOI: 10.1002/spe