

Enhanced Server Fault-Tolerance for Improved User Experience

Manish Marwah
Hewlett-Packard Laboratories,
Palo Alto, CA 94304

Shivakant Mishra
Department of Computer Science,
University of Colorado, Boulder, CO 80309

Christof Fetzer
Department of Computer Science,
TU-Dresden, Dresden, Germany D-01062

Abstract

Interactive applications such as email, calendar, and maps are migrating from local desktop machines to data centers due to the many advantages offered by such a computing environment. Furthermore, this trend is creating a marked increase in the deployment of servers at data centers. To ride the price/performance curves for CPU, memory and other hardware, inexpensive commodity machines are the most cost effective choices for a data center. However, due to low availability numbers of these machines, the probability of server failures is relatively high. Server failures can in turn cause service outages, degrade user experience and eventually result in lost revenue for businesses. We propose a TCP splice-based Web server architecture that seamlessly tolerates both Web proxy and backend server failures. The client TCP connection and sessions are preserved, and failover to alternate servers in case of server failures is fast and client transparent. The architecture provides support for both deterministic and non-deterministic server applications. A prototype of this architecture has been implemented in Linux, and the paper presents detailed performance results for a PHP-based webmail application deployed over this architecture.

1 Introduction

In recent years, computing applications and services are moving away from local desktop machines to remote data centers. This computing paradigm is attractive for a number of reasons: (1) It frees users from the issues and costs related to installing, maintaining and upgrading local software applications; (2) It allows easy access to applications and data from any location (using Internet connectivity); (3) It facilitates sharing and collaboration among multiple users who are geographically separated; and (4) It simplifies sending critical client software updates such as bug and security fixes to the clients (client scripts can be downloaded by the browser each time they are used). In the future, more applications are likely to migrate to remote data centers, effectively *remote desktops* that people can access via thin clients.

In order to ride the performance/cost curve for CPU, memory and other hardware, inexpensive commodity machines are most cost effective for a data center. However,

their availability numbers are low (about three nines). Thus use of commodity machines, rather than customized, hardened machines, leads to more server failures and service outages which in turn degrades user experience and results in lost revenue for businesses. For example, if a user is browsing a map service like MSN, Google or Yahoo maps, a server failure leading to an outage of *more than a few seconds* is detectable by a user and hence degrades user experience. However, *if server failures can be seamlessly handled, the low availability numbers of a server is not a problem.*

Many emerging Web applications are highly interactive (e.g., map browsing services) or even real time (e.g., stock market ticker). Other applications – such as word processing and spreadsheets – that have traditionally resided on a desktop are beginning to be hosted at a remote data center. In order to ensure seamless user experience, these applications put greater fault-tolerance demands on data centers. At present, server failures in Web server farms are typically handled as follows: 1) The client detects a server failure (usually by noticing an absence of response for some period of time); 2) The client reissues the request; 3) The re-sent request likely reaches a working server; 4) The new server handles the request. This procedure can easily take tens of seconds if not more. It is clear that these traditional mechanisms for handling server failures are no longer acceptable. In particular, application response times have a direct impact on user experience. It was recently reported [8] that Google has re-architected parts of its Gmail application in order to make the application faster – “[we are] profiling all parts of the application, shaving milliseconds off wherever we can”. While this would clearly provide a better user experience during normal operation, it does not address the problems of prolonged response times in case of server failures. In order to minimize the impact of low availability commodity servers, server failure recovery must be performed fast such that it is seamless to the user.

In this paper, we describe the design and implementation of a Web server architecture that provides improved user experience. In particular, it seamlessly tolerates failures of intermediate proxies that perform content-based routing as well as backend servers that process client requests. Furthermore, it provides support for handling non determinism in server applications during server failures. To provide improved user experience, this architecture incorporates the fol-

lowing important features: (1) Proxy or server failure detection is performed locally at the server end that is completely transparent to a client. This ensures a fast failure detection, in particular when a client is connecting over a wide area network (WAN). Fast failure detection by a client over a WAN is problematic as it leads to increased traffic and is prone to false positives. (2) Failover is significantly faster – at most a few seconds, something a client will consider a minor network glitch. (3) All client sessions and states are preserved during failover resulting again in a faster and seamless recovery.

The complete system architecture described in this paper is based on some of our earlier work on enhancements to TCP splicing mechanisms [11] and systems architectures for transactional network interfaces [12]. There are three unique contributions of this paper. First, the TCP splicing mechanism is adapted for seamless backend server failover. It allows for transparently redirecting current and future client requests to an alternate backend server in case of original backend server failures. Second, concepts of request transactionalization, tagging and logging have been introduced and assimilated to provide support for fast failover and seamless recovery. Finally, a prototype of the complete system architecture has been built and experimented with in both LAN and WAN (PlanetLab) settings. Fast failover and seamless recovery are demonstrated by deploying a real-world application (RoundCube Webmail [16], an open-source webmail client) over this architecture.

The rest of this paper is organized as follows. Section 2 describes some of our earlier work on which this work is built as well as some related work. Section 3 provides a high-level description of our complete system architecture. Section 4 describes important details including TCP re-splicing, transactionalization and tagging, and recovery mechanisms. Section 5 describes some salient features of our prototype implementation. Section 6 describes the details of RoundCube Webmail deployment over our architecture, and the performance measured from this deployment under many different operating scenarios. Finally, Section 7 concludes the paper and discusses some future work.

2 Background

2.1 TCP splice

Web proxies are exceedingly used in Web server architectures for implementation of layer 7 (or content-aware) routing, security policies, network management policies, usage accounting, and Web content caching. An application level Web proxy is inefficient since relaying data from a client to a server involves transferring data between kernel-space and user-space that results in additional context switches. TCP Splice was proposed [9, 17] to enhance the performance of Web proxies. It allows a proxy to relay data between a client and a server by manipulating packet header information entirely in the kernel. This makes the latency and computational cost at a proxy only a few times more expensive than that of IP forwarding. There is no buffering required at the proxy that performs the splicing, and, furthermore, the end-to-end semantics of a TCP connection are preserved between the client and the server. Advantages of TCP splicing in Web server architectures are further described in [15, 14, 5]. The

following steps are required in establishing a TCP splice:

- A client connects to a proxy. The proxy accepts the connection and receives the client request.
- The proxy performs authentication and other functions as configured by the administrator, and then performs layer 7 routing to select a backend server. It creates a new TCP connection to the selected server.
- The proxy sends the client request to the server and “splices” the client–proxy and the proxy–server TCP connections.
- After the two TCP connections are spliced, the proxy acts as a relay — the packets coming from the client are sent on to the server, after appropriate (in-kernel) modification of the header that makes the server believe that those packets are part of the original proxy–server TCP connection); similarly, the packets received from the server are relayed to the client after appropriate (in-kernel) header modifications.

2.2 Enhancements to TCP splice

The TCP splice mechanism described above suffers from two major drawbacks: (1) All traffic between a client and a server (both directions) must pass through a proxy, thus making the proxy scalability and performance bottlenecks; and (2) this architecture is not fault-tolerant; if a proxy fails, all the spliced TCP connections hosted on it fail as well, and clients have to re-establish their HTTP connections and re-issue failed requests. This would be true even in the presence of a backup proxy.

In order to address the scalability/performance bottlenecks and fault-tolerance issues, we proposed two important enhancements to the TCP splice mechanism [11]: (1) *Replicated TCP splice*: The splice state information is replicated on multiple proxies allowing one TCP connection to use multiple proxies. This distributed architecture provides both increased scalability and fault-tolerance; in fact, proxy fault-tolerance becomes trivial to implement as it only involves detecting that a proxy has failed and then ceasing to send packets to it. (2) *Split TCP splice*: The TCP splice functionality is split into two unidirectional splices with packets in the two directions being spliced at different machines. The packets destined to the server are spliced at a proxy as usual, however, response packets are spliced at the server and thus bypass the proxy. Splicing state information is copied to the server in order to achieve this. This further improves the scalability of the system particularly in cases where the response is large.

2.3 Related work

Our architecture is most related to FT-TCP [2, 19], ST-TCP [10], Backdoors [4, 18] and other similar systems. Unlike our architecture, FT-TCP and ST-TCP use an active backup which processes all requests sent to the primary server. Furthermore, the applications are required to be deterministic. Our architecture provides greater scalability by not requiring a dedicated backup and non-determinism is handled.

Backdoors [4, 18] requires a specialized NIC capable of performing remote direct memory access (RDMA). This allows a backup to read state information from a failed primary. Thus, Backdoors does not work if the failure impairs the primary memory, or, access to it. Our architecture can tolerate any server failure and no special hardware is required. Furthermore, Backdoors requires kernel modifications on primary and backup machines. Although kernel modules are needed on logger and proxy machines, we do not require kernel modification in backend server machines.

3 System architecture: An Overview

Figure 1 illustrates five logical components of our Web server architecture. Note that these are the functional components of the system. In an actual instantiation of this architecture, some of these components can be resident on the same machine. The five logical components are: (1) Stateless load balancers: Distribute incoming client requests to the proxies; (2) Proxies: Perform layer 7 routing, TCP splicing, and, re-splicing during recovery; (3) Backend servers: Process client requests, send back responses, and, asynchronously send application session state information to alternate backend servers; (4) Loggers: Transparently log traffic, parse requests and responses into tagged transactions, detect failure of backend servers, and assist in backend server recovery; and (5) Auxiliary servers: Additional servers that backend servers may contact for processing client requests.

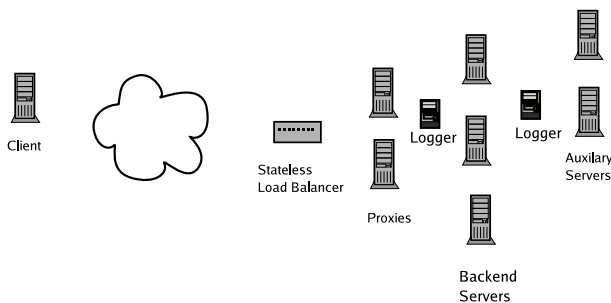


Figure 1: Components of our Web Server Architecture.

Stateless load balancers distribute incoming client packets among the proxies. As its name suggests, a load balancer is completely stateless and a packet is distributed regardless of the TCP connection to which it may belong. This also implies that load balancer fault-tolerance is simple to implement since there is no state to synchronize. The load balancer could also be co-resident at a layer 2 switch or an IP router.

For new connections, a proxy performs layer 7 routing and TCP splicing, and replicates the TCP splice among all proxies. Once a TCP splice has been replicated, subsequent client requests can be handled (in-kernel header transformation and forwarding to the appropriate backend server) by any proxy in the proxy stage. A proxy failure is trivial to handle. The recovery action comprises a load balancer detecting that a proxy has failed and ceasing thereafter to send any packets to that proxy [11]. In fact, multiple proxy failures are handled similarly. In this paper, we extend the role of proxies to assist in recovery from backend server failure by participating

in state synchronization of the alternate backend server and finally re-splicing the client TCP connection to that server.

A client request is handled at a backend server. A backend server can itself process a number of client requests. However, it may need to contact one or more additional servers in some instances for further processing. For example, a backend server may need to access a shared external database, or an email store. We refer to these additional requests as auxiliary (aux for short) requests, and the servers that handle these requests as aux servers. For certain client requests, a backend server may need to issue multiple aux requests. In order to correlate session state information and aux requests/responses with the appropriate transaction, a unique transaction ID is assigned to each transaction. This ID is computed from the client IP address and port number of the TCP connection and the ordinality of the transaction on that connection.

To facilitate seamless recovery, there are two points in the system where IP packets are logged — one is between a proxy and a backend server, where client requests and corresponding server responses are logged (front-end logger); the other is between a backend server and an aux server (aux logger), where aux requests and corresponding aux server responses are logged. Although two different loggers are shown in Figure 1, a single physical logger can be used for logging at both of these locations. In addition to logging IP packets, a logger performs a number of important functions to facilitate seamless recovery from backend server failures:

1. It parses all client requests and server responses into transactions.
2. It assigns tags to these transactions. These tags classify a transaction as deterministic/non-deterministic, stateful/non-stateful, etc.
3. It determines the mapping of these transactions to the TCP sequence numbers.

During normal (failure-free) operation, a client request is spliced at one of the proxies and dispatched to a backend server. This splice is also replicated at multiple proxies, so that different client requests can pass through any one of these proxies. The failure of a proxy is simply tolerated by detecting the proxy failure and not sending any subsequent requests to that proxy. Tolerating proxy failures and the related impact on performance has been discussed in detail in [11]. The backend server processes the client request (may involve zero or more aux requests) and sends the response back. The response may be sent via a proxy (which performs the splicing), or the backend server may itself perform the return half of the splice (split TCP splice) and send it directly to the client. In either case, the response is transparently logged at the front-end logger (which is an IP hop on the packet's return path). At the end of processing a request, the backend server asynchronously pushes application session state information related to that request to an alternate backend server. Note that the alternate server is not a dedicated backup server and could be providing service to other clients at the same time.

If a backend server fails in the middle of a transaction, that transaction is re-started at an alternate backend server

where the server application is already running. In addition, the original TCP connection is un-spliced at the proxies; any in-flight bytes saved at the front-end logger but not received by the client are re-transmitted; and finally the original client connection is re-spliced with the “new” backend server.

For known non-deterministic transactions, a backend server either saves the response at an alternate server before responding to the client, or synchronizes session state such that the alternate backend server can regenerate the same response to the request. Unforeseen non-deterministic conditions are rapidly detected by the system and the client is appropriately informed about them.

3.1 Logging, Transactionalization and Tagging

All client bytes destined for a backend server pass through the front-end logger. This logger is just an additional IP hop. It does not modify the packets in any way. In addition to saving the client bytes, the logger groups them into transactions that serves the following purpose:

- Transactionalization is used to give structure to a TCP byte stream, so that during failure recovery, an application can be re-started on an alternate (new) backend server *at a transaction boundary*.

The front-end logger uses an application-specific configuration file to determine the start and end of requests and responses. At present, we assume that each client request/response pair is a separate transaction. For example, if the application is an HTTP server, a simple approach is to treat each client request (e.g., a GET or a POST request) as a separate transaction. If the server application is a command shell, each request (commands separated by a newline or a semicolon) could be considered a separate transaction. For many applications, grouping of multiple requests/responses into a transaction is useful. However, that is outside the scope of this paper.

A mapping between transaction boundaries at the application layer and the TCP sequence number-space is required for seamless migration of the TCP connection to an alternate backend server in the event of a backend server failure. This mapping (as sequence number offsets from the initial sequence number (ISN) of the connection) is also saved at the front-end logger.

When a client request is transactionalized, the transaction is also tagged with attributes. These attributes are useful during recovery in determining quickly how a particular transaction is to be handled at the alternate backend server, e.g., does it have to be replayed? Some common tags are described below:

- **Deterministic/non-deterministic:** This tag indicates whether the transaction is deterministic or not. A deterministic transaction is one which would produce exactly the same response and cause exactly the same side effects when it is replayed on an alternate backend server. In other words, given the current state of an application and an input, only one output can be produced. An example of a deterministic transaction is the UNIX command `ls`. If the sequence of commands that are run on a backend server are replayed on an alternate backend server, `ls` would produce the same result. An example of a non-deterministic command is `date`.

- **Read-only/update:** This tag indicates whether a transaction changes the state of a backend server application in any way. Read requests usually do not have any side-effects. On the other hand, write requests change the state of the server and hence have side-effects. The Unix command `date`, for example, is read-only; however, `setenv DISPLAY remotemachine:0` results in an update of the state. On failover, a read-only transaction need not be replayed at an alternate backend if the response has already been generated and has either reached the client or is saved at the front-end logger. However, this is not true for update transactions. Consider, for example, a deterministic update transaction. If a backend server crashes after this transaction has been processed (reply sent to client) but before the corresponding state information is sent to an alternate server, then this transaction must be replayed at the alternate server.

- **Idempotent/non-idempotent:** An idempotent transaction is one that produces the same output and the same side effects whether it is processed once or multiple times. For instance, setting an environment variable (`setenv EDITOR emacs`) is an idempotent transaction. Non-idempotent transactions, on the other hand, must be executed only once, e.g., appending a value to an existing environment variable (`setenv PATH ${PATH}:/usr/sbin`). Care must be taken that non-idempotent transactions are not replayed on the alternate backend server if the corresponding state information has already been incorporated in the application session state.

Assignment of tags is application specific. For each application the likely client requests and the corresponding tags need to be specified in a configuration file. For each incoming request, the logger tries to match it with an internal table (constructed from the user specification). If a match is found, the corresponding information is used to tag the request. Otherwise, a default, conservative tag assignment is made to that request.

Large files transferred by a backend server to a client are not saved at a logger. Instead, file location information is saved. An alternate design could be to save a cryptographic hash (e.g., MD5) of the file and then use a mapping service to locate the file if needed.

3.2 Synchronization and Re-splicing

The front-end logger that saves requests and responses for a backend server also monitors that backend server for failures. Failure recovery consists of synchronizing the client TCP connection state, re-splicing the TCP connection, and synchronizing the application state on the alternate backend server. Synchronization of the client TCP connection state and application state is done using two key pieces of information: (1) the *last client ack* saved at the logger; and (2) the *last server byte* saved at the logger.

The last client ack indicates the next server byte that the client expects, i.e. it is guaranteed that all prior bytes have been successfully received by the client. However, server bytes between the last client ack and the last server byte

logged at the logger may not have been received at the client. So, the logger starts re-sending bytes starting from the last client ack. While it is possible that there are in-flight server bytes or acks, re-sending these bytes is harmless.

Synchronization of the application at the alternate backend server involves making sure that all the application session state information sent by the original backend server before failing has been applied at the alternate backend server. If the alternate backend server has lagged behind, some transactions may need to be replayed during recovery. In fact, only update transactions that change application state are replayed.

Once this synchronization is complete, the original client-proxy connection is respliced to the proxy-alternate backend server connection.

An issue during recovery is that if some transactions are re-run, they may generate auxiliary requests that may not be idempotent. Since, these requests have already run at the aux servers before the backend server failed, it is important to ensure that they are not re-sent to the aux servers during recovery. To address this, a second logger (aux logger) saves all auxiliary requests and corresponding responses. If a transaction is replayed at the alternate backend server during recovery and generates an auxiliary request, it is matched with the stored requests at the logger. The corresponding logged response is then returned without the participation of the corresponding auxiliary server.

3.3 Application Support

Our architecture requires some (minimal) support from the application to recover from server failures. In particular, the backend server application needs to transfer per transaction state information to an alternate backend server running that application. Also, this state information must have the granularity of a transaction and be applied to an already running application on the alternate backend server. Furthermore, for efficiency considerations, it is preferable that the application maintains only session state and that long-term persistent state is saved in auxiliary databases. Fortunately, this is also the usual industry practice.

Another requirement for the application is the inclusion of the transaction ID of a request within any auxiliary requests generated. Such an ID allows auxiliary requests to be correlated to the corresponding transactions. For example, for auxiliary requests to an IMAP server, a transaction ID can be part of the tag used with each IMAP command.

3.4 Non-determinism

Non-determinism implies that a request may produce a different response each time it is processed. In the context of synchronizing TCP connection state, non-determinism is a problem since the backend server may crash when only a partial response has been sent to the client. Unless the alternate backend server can regenerate an identical response, it is not possible to provide the rest of the response to the client and preserve the TCP connection state during recovery.

In our architecture, we address the issue of non-deterministic transactions in two ways. First, if it is known in advance that a particular transaction is non-deterministic (via non-deterministic tag), the application makes sure that before it starts sending a response, one of the following is true: (1)

the entire response has been saved at the alternate backend server, or (2) enough state information has been copied to the alternate backend server so that it can produce a deterministic response to that request.

However, it may be hard to identify all instances of non-determinism in an application in advance. This is because there might be some error conditions, or some uncommon user actions – not previously tested – that may produce non-deterministic responses. An important feature of our architecture is that it can detect such conditions arising from unforeseen sources of non-determinism. This is done by comparing the response bytes produced by the alternate backend server with the partial response – saved at the logger – produced by the original backend server before failing. If these do not match, the transaction is clearly non-deterministic. When such a situation is detected, the proxy sends a reset on the client connection, terminating it immediately. This would cause the client to reconnect and re-issue the request. Note that although not ideal, this approach is still better than a server simply failing, since the client is immediately notified that it needs to re-establish its TCP connection. Without this notification, it can take tens of seconds or more for a client to detect a server crash failure.

3.5 Adaptive Failure Detection

A backend server failure detector resides on the logger that is responsible for recording requests to and responses from that server. A two-pronged, adaptive server failure detection mechanism, with different approaches for times of activity and inactivity, is used. When a server is processing requests, it is declared failed if it does not respond to a request within a timeout. This timeout is dynamically computed by the detector as it observes requests and responses. For each kind of request, the detector maintains two timeouts based on the moving average of the following two measurements: (1) the time difference between receiving the entire request and the start of the response; (2) the time difference between the start and end of the response. Using two timeouts allows failure detection to be more fine grained than using one timeout value based on receipt of request to the end of response. Note that no heartbeats are used in this mechanism and failure detection is fine grained. We feel this approach has three main advantages over using a heartbeat mechanism: (1) there is no network or processing overhead of heartbeats; (2) the mechanism is adaptive and depends on the average responsiveness of the system rather than a fixed heartbeat interval value; and (3) the system designer does not have to pick a heartbeat interval value.

A low frequency heartbeat is used during idle periods. This is useful since the system is likely to be repaired before the next request comes in. Furthermore, the low frequency (once every few seconds) ensures that it does not put any detectable load on the system.

4 Detailed Design

4.1 Normal Operation

Steps required for handling a client request are as shown in Figure 2. Here it is assumed that the client has already established a TCP connection spliced by a proxy to a backend server. The proxy has also replicated the splicing state

information to other proxies so that any proxy can forward subsequent client requests [11].

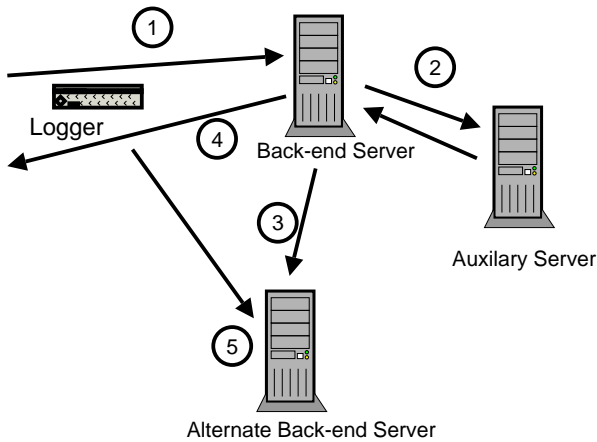


Figure 2: Sequence of steps required in handling a client request.

1. A client request is received by the backend server.
2. The backend server sends out any aux requests required for processing the client request and waits for the corresponding responses.
3. If the client request changes application state (“update” request), the updated state information is sent asynchronously to an alternate backend server. Note that because of asynchronous nature of this communication, the alternate backend server can lag behind the backend server. So, if the transaction is non-deterministic, this state information is sent synchronously.
4. The backend server sends a response to the client request.
5. When the logger receives the entire response, it informs the alternate backend server so that it applies that state information to the local application. Notice that the alternate backend server applies this state information only after the backend server has sent out the response and the response has been completely logged. Note that at the user level on the backend server, it is hard to determine when the response has been completely sent (since it may remain in the TCP send buffers for some time).

4.2 Terminology

We now define some terms that are used in the failure recovery process described in the next section. T_i refers to the unique transaction ID assigned to each transaction. It is a 64 bit long ID consisting of client IP address, client port number and the transaction number.

T_L Last transaction with response fully saved at the logger, i.e., the backend server crashed before the completion of transaction $T_L + 1$.

p Number of response bytes (zero or more) of transaction $T_L + 1$ saved at the logger.

T_S Last transaction with state information applied to the application at the alternate backend server. Since the alternate backend server waits for the logger to completely receive a response before applying the corresponding state information, $T_S \leq T_L$.

T_A Last transaction whose state information is available at the alternate backend server. Clearly, $T_A \geq T_S$. State associated with transactions $(T_S, T_A]$ is available at the alternate backend server, but has not yet been applied to the application.

T_R First transaction that is run on the alternate backend server during recovery.

T_{SP} First transaction that is sent over the re-spliced client and alternate backend server connection. Clearly, $T_{SP} \geq T_R$.

Ack_{CL} Last client ack saved at the logger.

Ack_{TL} Ack corresponding to the last response byte in transaction T_L .

4.3 Failure Recovery

The following steps are involved in the failure recovery process after a backend server has crashed. Note that although presented sequentially here for clarity, a number of these steps occur concurrently.

- Logger detects a backend server failure; determines T_L and shares this information with the alternate backend server.
- Proxy un-splices the client TCP connection with the failed backend server; signals other proxies to do the same.
- Alternate backend server determines T_S and T_A ; shares this information with the logger.
- **Synchronizing the client TCP connection.** Ack_{CL} is the last ack received from the client. Bytes are sent/re-sent to the client from this point. Bytes until the end of transaction T_L and p bytes of transaction $T_L + 1$ are already available at the logger. Therefore, if the following equation is true, the proxy temporarily re-splices the client connection with a new connection to the logger in order to send out these bytes.

$$Ack_{CL} \leq Ack_{TL} + p \quad (1)$$

Note that Equation 1 will very likely be an equality unless there is packet loss. Usually the time taken to detect failure – even if it is a second or less – is enough for the client ack of the last packet sent to be received by the logger. The client connection is re-spliced to the alternate backend server at transaction T_{SP} , which is,

$$T_{SP} = T_L + 1 \quad (2)$$

Since p bytes of $T_L + 1$ are already sent, the re-splicing is performed such that the first p bytes of the transaction are locally received at the proxy and thereafter the bytes are spliced to the client connection. Similarly, the splice point for bytes from the client to the server is also determined based on the last client bytes that is available on the logger.

- **Synchronizing the alternate backend server.** Although the client only needs response bytes starting from $T_L + 1$, the application state on the alternate server may not be updated until T_L . This could be due to two reasons: (1) the asynchronous state updates from the backend server lagged behind; or (2) signals from the logger – which cause an alternate server to apply the corresponding transaction associated state update – lagged behind. The alternate backend server is first updated till T_S . It applies state information associated with (T_S, T_R) and starts executing transactions at T_R , which is determined as follows.

$$T_R = \begin{cases} T_A + 1 & \text{if } T_L \geq T_A \\ T_L + 1 & \text{otherwise} \end{cases} \quad (3)$$

Note that replay of transactions $[T_R, T_L]$ is done intelligently; read-only transactions are not replayed. The client requests for $[T_R, T_L]$ and potentially $T_L + 1$ are supplied by the logger on the proxy–alternate server connection. Note that if a request is substantial in size, for instance, it is an HTTP POST request to transfer a large file, the proxy can splice the logger–proxy and proxy–alternate server connections.

- As mentioned earlier, the aux logger saves any aux requests and responses; an aux request carries a unique transaction ID which can be used to correlate it to a particular transaction. For the replay of transactions $[T_R, T_L]$ and potentially partial replay of $T_L + 1$, any aux requests are responded to by responses cached at the aux logger.

Note that in practice, the two loggers, backend server and alternate backend server most likely reside on the same LAN. Hence, $T_L = T_S = T_A$ is the most likely scenario, in which case only one transaction, $T_L + 1$, is replayed.

5 Implementation

We implemented a prototype of our server fault-tolerance architecture in Linux. We had earlier made some enhancements to TCP splice [11] to make it distributed and fault-tolerant. We further extended the TCP splicing functionality to perform re-splicing. This is implemented as a Linux kernel module and installed at a proxy. We enhanced our logger [12] to transparently log TCP connections and make the logged bytes available to a user-space transactionalizer and tagger. Finally, a recovery manager that resides at a proxy was added to coordinate recovery.

Netfilter. We made extensive use of netfilter [13] in both the kernel modules: logging module (`logmod`) and TCP splicing module (`tcpspmod`). Netfilter adds a set of hooks

along the path of a packet’s traversal through the Linux network stack. It allows kernel modules to register callback (CB) functions at these hooks. These hooks intercept packets and invoke any CB function that may be registered with that hook. After processing a packet, a CB function can decide to inject it back along its regular path, or steal it from the stack and send it elsewhere, or even drop it.

Logging kernel module. The logging module uses user-space memory that is mapped into the kernel. A user process sends a user-space memory pointer to the logging module using a system call. The kernel module calls `get_user_pages()` to map that memory into kernel-space memory pages. This allows it to log TCP segments on to memory that becomes visible to a user process without any kernel-to-user space copying. The module needs to take special care to detect and correctly log re-transmissions and out of sequence packets. Furthermore, since both the kernel module and a user-space process are accessing the same piece of memory, appropriate synchronization mechanisms are used to avoid race conditions. Since the memory allocated for logging is limited, the log wraps around on reaching the end of allocated space.

User-space Transactionalizer & Tagger The transactionalizer and tagger interacts with the logging module to obtain access to the memory where TCP stream data is being logged. Using application specific information, it *eagerly* parses the byte stream into transactions. Note that although transaction information is only needed if there is a failure, parsing the stream *lazily* – as needed on failure – is problematic since the log may wrap around. Each transaction is given a transaction ID and tagged. Tagging requires application specific knowledge as discussed in Sections 3.1. It also maintains a mapping of the TCP sequence number offsets to Transaction IDs. Furthermore, it communicates with the recovery manager at the proxy and with the alternate backend server during failure recovery.

Recovery manager. The recovery manager is a user-level process that resides at a proxy and is responsible for coordinating the failure recovery once a backend server failure is detected. It instructs the TCP splicing module to suspend the splice to the failed backend. It communicates with the transactionalizer and tagger to obtain the appropriate offsets when the backend crashes. Furthermore, it supplies the TCP splicing module with those offsets to perform re-splicing.

6 Experiments and Performance Evaluation

To provide a proof of concept, we demonstrate our architecture on a real-life Web mail application called Roundcube webmail [16], an open-source webmail application written in PHP programming language. Hosted at a data center, this application provides service similar to that provided by Google Mail, Yahoo Mail, or Microsoft’s Hotmail. Users can connect to it via Web browsers. Roundcube Web server uses IMAP [6] to connect to the email store servers. Compared to traditional webmail clients, Roundcube and other similar AJAX-based [1] Web applications have a more responsive user-interface.

We conducted a series of experiments to evaluate the efficacy of our architecture in terms of providing an improved user experience during a server crash failure. The main goals of the experiments were: (1) Measure failover times with

our architecture; (2) Measure the overhead of our architecture during normal operation; (3) Compare failover times obtained with our architecture to those obtained with commonly used current server fault-tolerance techniques; and (4) Evaluate our system with clients connected over networks with diverse characteristics.

We conducted experiments in both LAN and WAN settings. The backend servers, proxy and logger that we used are attached to the same LAN on the campus network of the University of Colorado at Boulder. To test under a LAN environment, a client application was installed on a machine connected to the same LAN. In order to run our experiments over diverse WAN links, we used PlanetLab [3] nodes as our testbed. We placed the client on three distinct sites: (1) WAN-MIT: a PlanetLab node at MIT (planetlab7.csail.mit.edu); (2) WAN-SG: a PlanetLab node in Singapore (planetlab3.singaren.net.sg); and (3) WAN-IN: a PlanetLab node in India (planetlab1.iitr.ernet.in). These locations were chosen because they provide a wide variety of round trip times (RTT) between the server in Colorado and its peer. The RTT is about 79 ms to the machine at MIT, and about 260 ms to the one in Singapore. The RTT to the node in India is extremely large at about 886 ms. Compared to these, the RTT for the LAN scenario is about 0.25 ms.

6.1 Experimental Setup

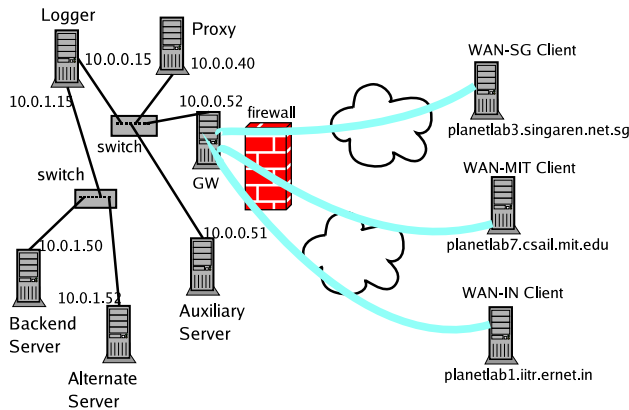


Figure 3: Experimental setup.

The experimental setup is shown in Figure 3. At the server end, the machines are attached to two Ethernet switches and span two subnets. The proxy, logger and the auxiliary server (which acts as an IMAP server) reside on 10.0.1.0/24 subnet. This subnet is connected to the public Internet through a GW machine. The logger and backend servers are part of the 10.0.0.0/24 subnet. Note that the logger is dually homed and is a gateway between the two subnets. This allows it to conveniently log packets between both the backend server and a client, and the backend server and the auxiliary server. Since the machines are on private subnets, the WAN clients connect to it through *ssh* tunnels between the client and the GW machine.

Action	GET				POST
	PHP script	File Downloads			
		CSS	JavaScript	images	
Login screen	1	1	2	4	0
Logging in	2	1	1	28	1
Reading msg	2	1	1	4	0
Sending msg	3	0	0	1	1

Table 1: Common actions in Roundcube and the corresponding HTTP GET and POST requests. For the GET requests, the number of times a PHP script is invoked at the server, and, the number of CSS, JavaScript and images files that are download by a Web browser are also listed.

6.2 Experiments

Users connect to webmail using HTTP through a Web browser. User actions such as logging in, reading and sending email are translated into HTTP GET and POST requests by the browser as shown in Table 1. For our experiments, we chose common actions that users are likely to take while checking their email. Furthermore, we chose both read-only and update actions.

We picked two actions that were used for all our experiments. The first is a simple one: displaying the login screen. The second is a more complex operation: it consists of a user logging in, composing an email, sending it out and, finally, logging out. We experimented with several other user actions as well and these two are well representative of the lot since they have a mix of read-only and update operations.

In order to be able to send these requests repeatedly, measure the time taken, and cause a failure when a request is in progress, we used a 'C' program instead of a browser as the client for our experiments. We performed both the above actions and used *ethereal* [7] to record the requests sent out by the browser and the responses received. To make sure that our program operated identically to a browser, we sent these recorded requests to the Web server. We also matched the responses received to the recorded ones to ensure correctness. Furthermore, the program parsed the received response header in order to correctly receive the body of the response. The webmail application assigns a session ID and sends it as a cookie when a user logs in. This ID needs to be sent with all subsequent requests in that session. This was another capability that we added to our client program.

Action 1: Displaying the login screen. This action consists of eight GET requests in all. About half are requests for images displayed on the login screen. One is an execution of a PHP script and others download JavaScript and CSS scripts. For our experiments, we assume that the images are already cached at the client and issue the remaining four GET requests. Each run consists of the client establishing a TCP connection on port 80 and repeating these four GET requests 30 times. (Actually they are repeated 31 times and the first set is ignored to minimize the impact of startup cost and cache misses.) We perform the experiments under four different network settings: with the client on the same LAN as other machines; and with the client at WAN-MIT, WAN-SG and WAN-IN. For each of these network settings, we per-

form experiments for four different scenarios: (1) no failure and without the infrastructure required for our architecture; (2) server failure and traditional server fault-tolerance support implying that the client detects server failure through a heartbeat mechanism and reissues the failed request (which then is processed by an alternate backend server); (3) no failure but with our architecture infrastructure deployed; and (4) server failure with our fault-tolerance mechanism in place. Furthermore, each run is repeated at least three times and the average taken.

Action 2: User email session. This action mirrors the following user interaction. It starts with the user logging in. The INBOX folder is displayed with new messages, if any. The user then hits 'Compose' and drafts an email. Finally, the user sends out the email and logs out. This action consists of tens of requests, a large percentage of which are GET requests to download icons and images. There are also two POST requests: (1) for logging in the user; and (2) for submitting the user's email to the Web server. Again, we assume that the images are already cached and do not issue those requests in our experiments. Each run consists of nine GET requests and two POST requests which are repeated three times, that is, three email sessions are created and three emails sent out. Again, the experiments are repeated for four different network conditions and for the four scenarios described above for Action 1.

6.3 Results and Discussion

The results of our experiments for Action 1 and Action 2 are summarized in Tables 2 and 3, respectively. They list the average times taken for a run (consisting of 30 sets of four requests for Action 1 and 3 sets of 11 requests for Action 2) under diverse network and architectural scenarios. The key measurements to note are the failover times which are differences between the time taken during a failure-free run and a run with a server crash failure. Since we are interested in the user experience during server failure recovery, a large failover time – leading to degraded user experience – is unacceptable. The failover times using our architecture are all under about 3 seconds, with the exception of WAN-IN which is discussed later. For Action 1, conservative failure detection parameters are used and failure detection times of 1-2 secs were observed. These can be made even shorter since the logger and backend server are on the same LAN. For Action 2, more aggressive values were used and failure detection times of around 500ms were observed. One trend to notice in the results for both Action 1 and Action 2 is that the failover times tend to increase with increase in RTT times between the client and the server. We believe this is so because, on failure, the proxy establishes a new TCP connection with the alternate backend server. For congestion avoidance, this new TCP connection performs slow start which takes longer for a larger RTT.

The failover times also depend on the exact point of occurrence of the failure. A server failure can occur: (1) in between two transactions; (2) in the middle of a request; (3) in between a request and its response; and (4) in the middle of a response. During our experiments, backend server failure is caused by disabling the server's network interface a random time interval after starting the client. Most times the failure occurs in between a request and its response, that

is, the request is received but no response is generated yet. It took many runs to find an instance where failure occurred in the middle of a response. Considering that most replies in our experiments are short, the failover time was not very different from the other cases, however, it did provide us with more confirmation that our recovery manager and TCP resplicing code are correctly implemented. We believe that for large replies, especially if a failure occurs towards the end of the response, our architecture will be very effective. In a few instances, we were able to cause a failure in between two transactions. This leads to slightly faster recovery as replay of a failed request is not required.

The failover times for a traditional architecture are also listed in the two tables. As described earlier, a traditional system is assumed to be not client transparent and the client detects failure using a heartbeat mechanism. The heartbeat values used for LAN, WAN-MIT, WAN-SG and WAN-IN were 1s, 5s, 10s and 20s respectively. Failure is declared if three heartbeats are missed and thus take between two and three times the heartbeat interval value. For traditional architecture, failure detection is a major part of the failover time, especially for WAN connections, since a very high frequency heartbeat is not practical. In our architecture, failure detection occurs locally at the server and server failures can be aggressively detected, irrespective of client location. Our architecture's overhead during normal operation is listed in the last column of the results tables. These values are low – with the maximum being 2.6% for Action 1. Although a bit higher for Action 2, the overhead values are still low.

The results for WAN-IN are peculiar and different from clients at other WAN locations: the average overhead due to our architecture for WAN-IN is negative; failover time is high for Action 1, but seems low for Action 2. We believe this is due to temporal variations in the network characteristics of the link while we were conducting the experiments. The RTT, in addition to being very high also has a large mean deviation of close to 100 ms, as measured using ping.

We encountered two potential instances of non-determinism while running our experiments: (1) a date field in the HTTP response header; and (2) a "keepalive" field in the HTTP header indicating the number of subsequent requests that can be sent on the same TCP connection. Both of these have simple fixes. The date field has a fixed length and thus does not cause any problems at the TCP layer. Furthermore, it would be an issue at the application layer only if the date is partially sent when a backend server fails. In practice, this is a very unlikely scenario since the date is in the first few bytes of the response header and is most likely to be sent atomically. In a pathological scenario, where this is not true, any non-determinism that occurs will be detected by our architecture. The keepalive field is a problem at the TCP layer since its size is not fixed. However, its impact at the application layer is inconsequential. Making the length of this field constant in the HTTP header will be a simple fix to this problem and remove the non-determinism at the TCP layer. In our experiments, we configured the HTTP server to not restrict the number of requests on a TCP connection and thus this field was absent from the HTTP header.

We also found that for all our experiments $T_L = T_S = T_A$

	Average Time Taken (sec)						
	Traditional Architecture			Our Architecture			Overhead during Normal Operation
	No-Failure	With Failure	Failover	No-Failure	With Failure	Failover	
LAN	3.34	6.44	3.10	3.37	5.10	1.73	0.03 (0.89%)
WAN-MIT	23.0	35.9	12.9	23.6	25.10	1.50	0.6 (2.6%)
WAN-SG	68.1	105.5	37.4	68.5	71.45	2.95	0.4 (0.58%)
WAN-IN	299.2	373.5	74.3	297.7	314.7	17.0	(1.5)

Table 2: Time taken for performing one run of Action 1: Connecting to the login screen.

	Average Time Taken (sec)						
	Traditional Architecture			Our Architecture			Overhead during Normal Operation
	No-Failure	With Failure	Failover	No-Failure	With Failure	Failover	
LAN	3.72	6.69	2.97	3.83	4.77	0.94	0.11 (2.9%)
WAN-MIT	7.34	20.8	13.46	7.50	8.95	1.45	0.16 (2.17%)
WAN-SG	15.74	45.8	30.06	16.9	20.41	3.51	1.16 (7.3%)
WAN-IN	70.84	128.78	57.94	73.91	72.78	1.13	3.07 (4.3%)

Table 3: Time taken for performing one run of Action 2: Logging in; drafting and sending an email; and, logging out.

(using terminology from Section 4.2). Furthermore, Ack_{CL} always corresponded to the last server bytes saved at the logger. From the log messages of the logger, we noticed that there were only a couple re-sent packets implying that very few packets were dropped or delayed during our experiments; and there were no out-of-order packets received.

7 Conclusions

Server fault-tolerance assumes great significance in the light of explosive growth in emerging Web-based applications hosted at data centers. If server failures can be seamlessly and client-transparently tolerated, businesses can deploy cost-effective, commodity servers at data centers. In this paper, we presented a TCP splice-based server fault-tolerance architecture particularly aimed at reducing failover times to provide improved user experience during server failure recovery. The main components of our architecture are logging, transactionalization and tagging of user requests and responses, connection synchronization and re-splicing. We also address non-determinism and use adaptive failure detection. We have implemented a prototype of our architecture in Linux and demonstrated its effectiveness by deploying it with a real-life webmail application. For our experiments, LAN and WAN (using PlanetLab nodes) clients were used to issue common webmail actions, backend server failure was caused in the middle of request processing, and the failover times were measured. The results showed that the failover time is at most a few seconds even for clients connected over a WAN in contrast to traditional server fault-tolerance techniques where such failure detection itself can take tens of seconds.

References

- [1] Ajax, <http://wikipedia.org/ajax>.
- [2] L. Alvisi, T. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov. Wrapping server-side TCP to mask connection failures. In *Proceedings of Infocom 2001*, April 2001.
- [3] A. C. Bavier, M. Bowman, B. N. Chun, D. E. Culler, S. Karlin, S. Muir, L. L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating systems support for planetary-scale network services. In *NSDI*, pages 253–266. USENIX, 2004.
- [4] A. Bohra, I. Neamtiu, P. Gallard, F. Sultan, and L. Iftode. Remote repair of operating system state using backdoors. In *Proceedings of The International Conference on Autonomic Computing (ICAC-04)*, 2004.
- [5] A. Cohen, S. Rangarajan, and J. H. Slye. On the performance of tcp splicing for url-aware redirection. In *USENIX Symposium on Internet Technologies and Systems*, 1999.
- [6] M. Crispin. Internet Message Access Protocol - Version 4, Rev1. Request For Comments 2060, Internet Engineering Task Force, 1996.
- [7] Ethernal, <http://ethereal.com>.
- [8] Gmail Blog, <http://gmailblog.blogspot.com/2007/10/code-changes-to-prepare-gmail-for.html>.
- [9] D. A. Maltz and P. Bhagwat. MSOCKS: An architecture for transport layer mobility. In *Proceedings of INFOCOMM '98*, March 1998.
- [10] M. Marwah, S. Mishra, and C. Fetzer. TCP server fault tolerance using connection migration to a backup server. In *Proceedings of IEEE Int. Conf. on Dependable Systems and Networks*, San Francisco, June 2003.
- [11] M. Marwah, S. Mishra, and C. Fetzer. Fault-tolerant and scalable tcp splice and web server architecture. In *SRDS*, pages 301–310. IEEE Computer Society, 2006.
- [12] M. Marwah, S. Mishra, and C. Fetzer. Systems architectures for transactional network interface. In *10th IEEE High Assurance Systems Engineering Symposium*, Dallas, TX, Nov. 2007.
- [13] Netfilter, <http://www.netfilter.org>.
- [14] M.-C. Rosu and D. Rosu. An evaluation of TCP splice benefits in web proxy servers. In *WWW*, pages 13–24, 2002.
- [15] M.-C. Rosu and D. Rosu. Kernel support for faster web proxies. In *USENIX Annual Technical Conference, General Track*, pages 225–238, 2003.
- [16] Roundcube webmail, <http://roundcube.net>.
- [17] O. Spatscheck, J. S. Hansen, J. H. Hartman, and L. L. Peterson. Optimizing TCP forwarder performance. *IEEE/ACM Transactions on Networking*, 8(2):146–157, 2000.
- [18] F. Sultan, A. Bohra, I. Neamtiu, and L. Iftode. Nonintrusive remote healing using backdoors. In *Proceedings of First Workshop on Algorithms and Architectures for Self-Managing Systems*, 2003.
- [19] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. Bressoud. Engineering fault tolerant TCP/IP services using FT-TCP. In *Proceedings of IEEE Int. Conf. on Dependable Systems and Networks*, San Francisco, June 2003.